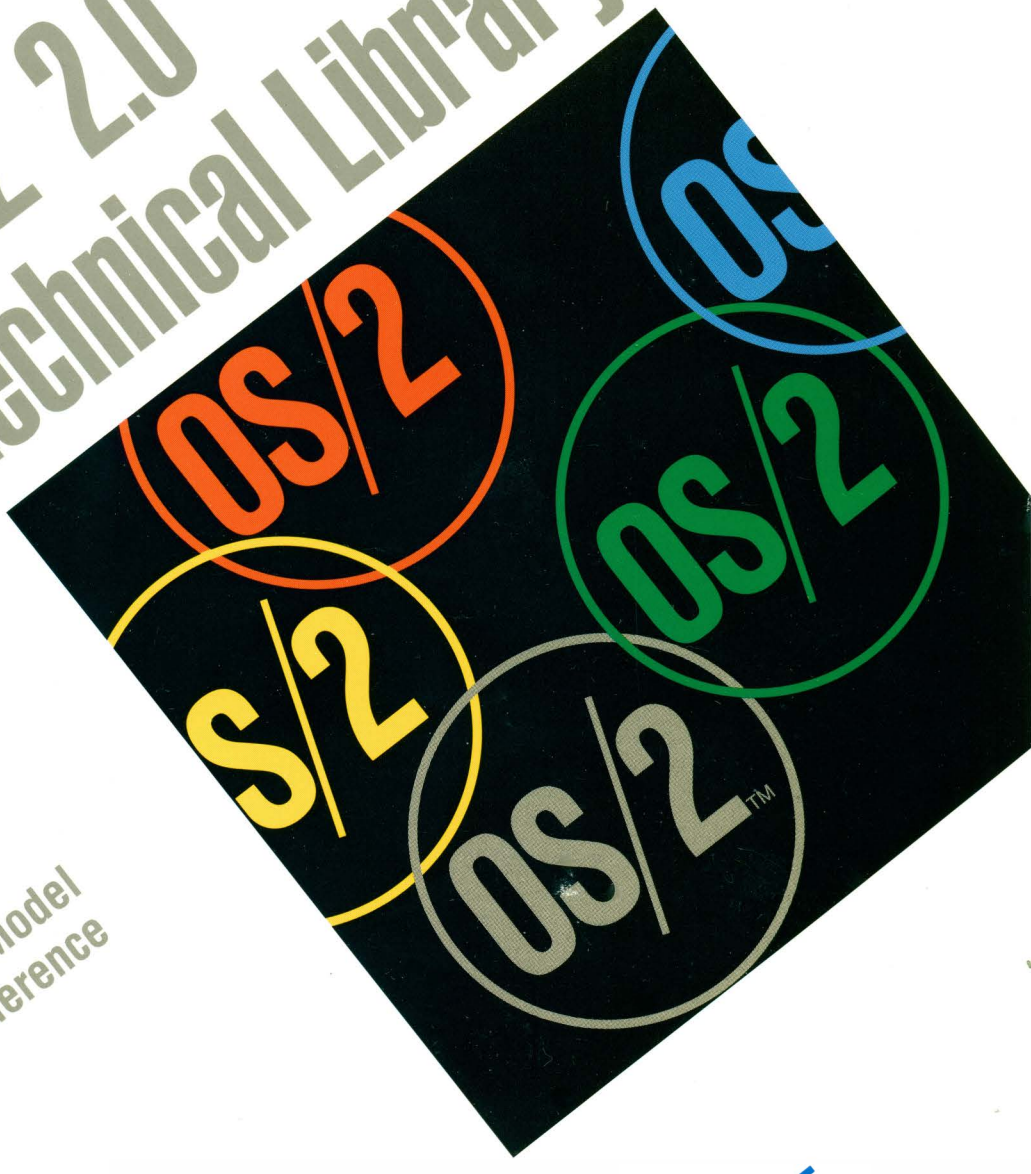
OS/2 2.0 Technical Library



System Object Model Guide and Reference

Version 2.00

IBM Programming Family

OS/2 2.0 Technical Library

System Object Model Guide and Reference

Version 2.00

IBM Programming Family

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page v.

First Edition (December 1991)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "© (your company name) (year) All Rights Reserved."

Contents

Notices v

About This Book vii
How This Book Is Organized vii
Who Should Read This Book vii

Chapter 1. Overview 1-1
Background 1-1
Introducing SOM 1-2
Using SOM 1-3
The SOM Run-Time Environment 1-4

Chapter 2. The SOM Development Process 2-1
Object Interface Definition Language 2-2
Include Section 2-3
Class Section 2-5
Release Order Section 2-8
Metaclass Section 2-9
Parent Class Section 2-10
Passthru Section 2-11
Data Section 2-12
Methods Section 2-14
SOM Compiler 2-17
Running the SOM Compiler 2-19
The SMINCLUDE Environment Variable 2-19
The SMEMIT Environment Variable 2-19
The SMTMP Environment Variable 2-19
SOM Compiler Command Syntax 2-20
Sample VECTOR.CSC file in OIDL 2-22

Chapter 3. SOM bindings for C 3-1
A Brief SOM Programming Primer 3-2
Using a Class and Its Objects 3-2
Defining a Class 3-4
Expansion of the SOM binding macros 3-5
API for Class Clients 3-6
<className>NewClass 3-6
<className>New 3-6
<className>Renew 3-6
_<className> 3-7
Method Macros 3-7
SOM_Resolve and SOM_ResolveNoCheck 3-8
Public Instance Variable Macros 3-9
API for Class Implementers 3-10
Implementation Conventions for Methods 3-10
<className>GetData Macro 3-10
Private Instance Variable Macros 3-11
Parent Method Macros 3-11
SOM_ParentResolve 3-11
API for General Usage 3-13
ID Manipulation 3-13
Debugging Facilities 3-14

Error-Handling Facilities 3-16
SOM_GetClass 3-16
SomEnvironmentNew 3-17

Chapter 4. Customization features 4-1
Memory-Management Functions 4-1
DLL Management Functions 4-2
Character Output Function 4-3
Error-Handling Function 4-4

Chapter 5. Classes Reference 5-1
SOMObject 5-1
SOMClass 5-2
SOMClassMgr 5-4

Chapter 6. Methods Reference 6-1
somAddStaticMethod 6-1
somCheckVersion 6-3
somClassFromId 6-5
somClassReady 6-6
somDescendedFrom 6-7
somDispatchX 6-9
somDumpSelf 6-11
somDumpSelfInt 6-12
somFindClass 6-13
somFindClsInFile 6-15
somFindMethod 6-17
somFindMethodOk 6-19
somFree 6-21
somGetApplyStub 6-22
somGetClass 6-24
somGetClassData 6-25
somGetClassMtab 6-26
somGetClassName 6-28
somGetInitFunction 6-29
somGetInstanceOffset 6-30
somGetInstancePartSize 6-32
somGetInstanceSize 6-34
somGetMethodOffset 6-36
somGetName 6-38
somGetNumMethods 6-39
somGetNumStaticMethods 6-40
somGetParent 6-41
somGetPClsMtab 6-42
somGetSize 6-44
somInit 6-45
somInitClass 6-47
somIsA 6-49
somIsInstanceOf 6-51
somLoadClassFile 6-53
somLocateClassFile 6-55
somMergeInto 6-56
somNew 6-58
somOverrideSMethod 6-59
somPrintSelf 6-60
somRegisterClass 6-61

somRenew 6-62
somRespondsTo 6-64
somSetClassData 6-66
somSupportsMethod 6-67
somUninit 6-69
somUnloadClassFile 6-71
somUnregisterClass 6-72

Appendix A. Error Codes A-1

Glossary X-1

Index X-3

Notices

The following terms are trademarks of the IBM Corporation in the United States and/or other countries:

Operating System/2
OS/2

The following term, denoted by a double asterisk (**) in this publication, is a trademark of another company:

Objective-C	The Stepstone Corporation

The term "ANSI C" used throughout this publication refers to American National Standard X3.159-1989.

About This Book

This book deals with the OS/2 System Object Model (SOM) and explains how C-language programmers can create programs that use SOM. Use this book as an overview of SOM, a guide to the tools associated with SOM, and later as a programming reference.

How This Book Is Organized

This book contains three distinct parts of two chapters each, an appendix, and a glossary.

1. The first part (chapters 1 and 2) introduces the reader to SOM, describes its major elements, and discusses the SOM software development process.
2. The second part (chapters 3 and 4) describes how SOM programming is accomplished using the C language.
3. The third part (chapters 5 and 6) contains reference information about the classes and methods supplied with SOM.

Who Should Read This Book

This book is for the professional C-language programmer.

The material on SOM is expressed in the commonly used terminology of object-oriented programming (OOP). A number of important OOP terms come from everyday English, but take on specialized meanings. These appear in the glossary at the back of the book. You may find it worth consulting if you discover yourself puzzling over the strange significance attached to an otherwise ordinary word.

This book assumes that you have a general familiarity with the basic notions of object-oriented programming and that you are an experienced C-language programmer. Practical experience with using an object-oriented programming language may also be helpful, but is not essential.

If you need a good introduction to object-oriented programming or a general survey of the many aspects of the topic, you might enjoy reading one of the following books:

- Booch, G, Object-Oriented Design with Applications, Benjamin/Cummings 1991, ISBN 0-8053-0091-0
- Budd, T, An Introduction to Object-Oriented Programming, Addison-Wesley 1991, ISBN 0-201-54709-0
- Cox, B, and Novobilski, A, Object-Oriented Programming, An Evolutionary Approach 2nd Edition, Addison-Wesley 1991, ISBN 0-201-54834-8

Chapter 1. Overview

Background

Among developers of workstation software object-oriented programming (or OOP) is increasingly recognized as an important new programming technology. It offers expanded opportunities for software reuse and extensibility, with improved programmer productivity when compared to conventional software development paradigms. Even so, penetration of object-oriented technology to major commercial software products, and in particular the operating-systems arena, has progressed slowly because of certain obstacles.

As with many new programming technologies, the early expressions of OOP concepts focused on the creation of new languages and toolkits, each designed to exploit some particular aspect. But new languages sometimes come at a price.

So-called "pure" object-oriented languages (such as Smalltalk), presume a complete run-time environment (sometimes known as a virtual machine) because their semantics represent a major departure from traditional "procedurally-oriented" system architectures. "Hybrid" languages such as C++, on the other hand, require less run-time support but sometimes result in tight bindings between programs that provide objects and the client programs that use them. Tight binding between object-providing programs and their clients often requires client programs to be recompiled whenever simple changes are made in the providing programs.

Because different languages and object-oriented toolkits emphasize different aspects of OOP, the utility of the resulting software is frequently limited in scope. A C++ programmer, for example, cannot easily use objects developed in Smalltalk, nor can a Smalltalk programmer make effective use of C++ objects. Objects and classes implemented in one language simply cannot be readily used from another. Unfortunately when this occurs one of the major benefits of OOP, the increased reuse of code, is severely curtailed. Object-oriented language and toolkit boundaries become, in effect, barriers to interoperability.

OOP encourages programmers to organize their software as sets of cooperating objects. Common functions typically are distributed over separate but interacting objects that each implement the specified function in a unique way appropriate to the individual object. This property is called *polymorphism* and frequently permits software to evolve with changes localized to particular objects. This style of organizing program content generally results in frequent interactions among cooperating objects, and motivates the need for a fast, efficient, light-weight mechanism for their expression. The need to keep objects loosely integrated so that they can be independently evolved and maintained, runs counter to this goal, and a good balance is difficult to achieve. C++, for example, tilts the scale in favor of efficiency with tight integration and frequent recompilations. Other languages, such as Smalltalk, exhibit less interdependence of objects, but at the cost of a single large image, or name space, for all objects.

For operating systems, these problems represent serious issues. The benefits of OOP can be indisputably appealing, and for state-of-the-art graphical user interfaces, OOP is probably essential. But a commitment to a single language by an

operating system might result in the perception of a closed system, and alienate some portion of the potential development community. Hence, adding objects, and programming interfaces to objects, to an operating system component, poses a number of challenges. It must be possible to access and modify the behavior of operating-system objects from a variety of languages. Equally important, the integration between operating-system objects and user-supplied objects must be as efficient as possible without requiring users to recompile their programs when the next version of the operating system appears. In the OS/2 software market, programs are generally provided in binary form and users do not have the capability to recompile them to accommodate a new release of the operating system.

The System Object Model (or SOM) is IBM's solution to this set of problems.

Introducing SOM

SOM is an object-oriented programming interface for building, exposing, and manipulating software objects. Unlike object models found in formal object-oriented programming languages, SOM is language-neutral. It doesn't require that the user of an object and the implementer of an object do their programming in the same language. This independence of implementation and client language extends even to the concept of subclassing. A user of a SOM class can create a subclass that may or may not be written in the same language as its superclass.

With SOM, the only detailed information available to clients about an object's implementation is that which the implementer has deliberately chosen to expose. SOM permits implementers to publish methods and selected instance variables whenever it is appropriate to do so, but once published, this information is considered to be a permanent part of an object's interface. Even so, SOM permits many implementation details to change without perturbing the object's interface. With a little careful consideration, the implementer of a SOM class can retain the ability to make evolutionary changes with each software release without requiring any recompilation of client programs.

As a rule of thumb, if you make changes to a SOM class that will not require source code changes in client programs, the binary versions of those programs also will not need to be recompiled. This is not true of all native object-oriented languages, and is one of the chief benefits you can expect from using SOM. Among the structural changes you can make to SOM classes (while retaining full, backward, binary compatibility) are:

- Adding new methods.
- Adding, changing, or deleting unpublished instance variables, provided that old methods can still be supported.
- Inserting new classes above your class in the inheritance hierarchy.
- Relocating methods upward in the class hierarchy.

In short, you can make the typical kinds of changes to an implementation that evolving software systems experience over time.

SOM is said to be "language-neutral" for three reasons. (1) All SOM interactions are built out of standard procedure calls. (2) SOM classes support several diverse mechanisms for method resolution that can be readily mapped into the semantics of a wide range of different commercial object-oriented programming languages. (3) The form of the SOM Application Programming Interface (API) can vary widely from

language to language as a result of special adaptations referred to as SOM "bindings."

The SOM method resolution mechanisms are:

- Offset resolution: Roughly equivalent to the C++ virtual function concept. Offset resolution implies a static scheme for typing objects, with polymorphism based strictly on class inheritance. It offers the best performance characteristics for SOM method resolution at a cost of some loss in flexibility. Methods accessible through offset resolution are called *static* methods because they are considered a fixed aspect of an object's interface.
- Name resolution: Similar to that employed by Objective-C** and Smalltalk. Name resolution supports untyped (sometimes called "dynamically" typed) access to objects, with polymorphism based on the actual protocols that objects honor. Name resolution offers you the opportunity to write code to manipulate objects with little or no awareness of the type or shape of the object when your code is compiled.
- Dispatch function resolution: A unique feature of SOM that permits method resolution to be based on arbitrary rules known only in the domain of the receiving object. Languages that require special entry or exit sequences, or local objects that represent distributed object domains are good candidates for using dispatch function resolution. This technique offers the highest degree of encapsulation for the implementation of an object, with some cost in performance.

In general, for each method defined in a SOM class, the implementer decides whether to support offset or name resolution. Dispatch function resolution is a property of an entire class rather than an individual method. If a particular form of dispatching mechanism is needed for the class, the implementer should override the dispatching methods provided in the SOMObject class.

Clients of SOM classes also can make choices about method resolution. If the implementer has selected offset resolution for a method, client programs can use any of the three SOM method-resolution techniques. If the implementer has selected name resolution for a method, client programs can use either name resolution or dispatch function resolution. And finally, implementers can choose to support only dispatch function resolution for all methods; in this case, client programs are similarly confined to use dispatch function resolution exclusively.

Using SOM

As a programmer you can use SOM in either of two basic ways.

1. If you program in a native object-oriented language (such as C++ or Smalltalk), you can use SOM as a standard way of importing existing classes, or exporting the classes that you have developed in the native language. This way you can access or subclass classes implemented in other languages, and offer your own classes for use from other languages without needing to invent a special interface for each "foreign" language.

   Exporting a native language class through SOM is done by first defining a corresponding SOM class as a representative of your class. Whenever an

** Trademark of The Stepstone Corporation.

instance of your class is created, the SOM class can be used to manufacture a proxy for your object. Users then interact with your object by using the methods of the SOM (proxy) object, which forwards them directly to your native object.

Importing a SOM class into your native language is similar, but in this case you define a native language class that is a proxy for the SOM class. Native language operations on your proxy class must in turn forward the operations to the SOM class.

The phrase "SOM bindings," in this context, is used to describe any automated process that assists in the creation of export or import proxies.

2. If you program in a language that has no object model of its own (such as the C language) you can use SOM directly to access existing classes or create new classes. Your language, in conjunction with the SOM bindings, effectively becomes a full object-oriented programming language. In this case, SOM objects instantiated from the classes you create are real objects in their own right, rather than proxies that represent other objects.

   In this context, the phrase "SOM bindings" is used to describe a set of macros or procedure calls (compatible with your language) that can be used directly to access and manipulate SOM objects.

The Developer's Toolkit for OS/2 2.0 (or Toolkit) supplies a complete set of SOM bindings for the C language. Vendors of other programming languages might also offer SOM bindings. Check with your language vendor about possible SOM support.

The SOM Run-Time Environment

One way to think about SOM is as a basic mechanism that can be used to express common object-oriented constructs, such as inheritance or method resolution. As such, SOM needs a simple run-time environment to hold essential data structures and manage their semantics. The SOM environment can be created automatically or explicitly within any process that uses it.

Because SOM is a basic mechanism, its run-time model is explained in terms of operations that occur on a single thread within a single process. Even so, all of the code in SOM is completely reentrant and permits concurrent execution by multiple threads. Multi-threaded programs can use OS/2* mutual exclusion semaphores to serialize updates to objects, but within the SOM run-time itself updates to global resources are automatically serialized. Complex object interactions that need to span process boundaries also can be constructed on top of SOM using standard OS/2 interprocess communication facilities, and encapsulated within particular SOM classes.

If you program to the basic "single-thread, single-process" model, you do not need to provide any serialization code. If you write multi-threaded programs that make use of SOM objects, you are responsible for providing synchronization and resource protection for any objects that may be shared across threads or processes. Method calls to shared objects can be protected with OS/2 semaphores.

One of the distinguishing characteristics of SOM is that SOM classes are real objects that play an active role in its run-time environment. When a SOM "class" object is constructed, many of its properties are established dynamically, including

* Trademark of the IBM Corporation

its relationships with other class objects. The ease with which evolutionary changes can occur in SOM classes can be attributed in a large part to the dynamic nature of the class construction process.

During SOM initialization, four objects are created, three of which are classes. The relationships among these SOM run-time objects are illustrated in Figure 1-1

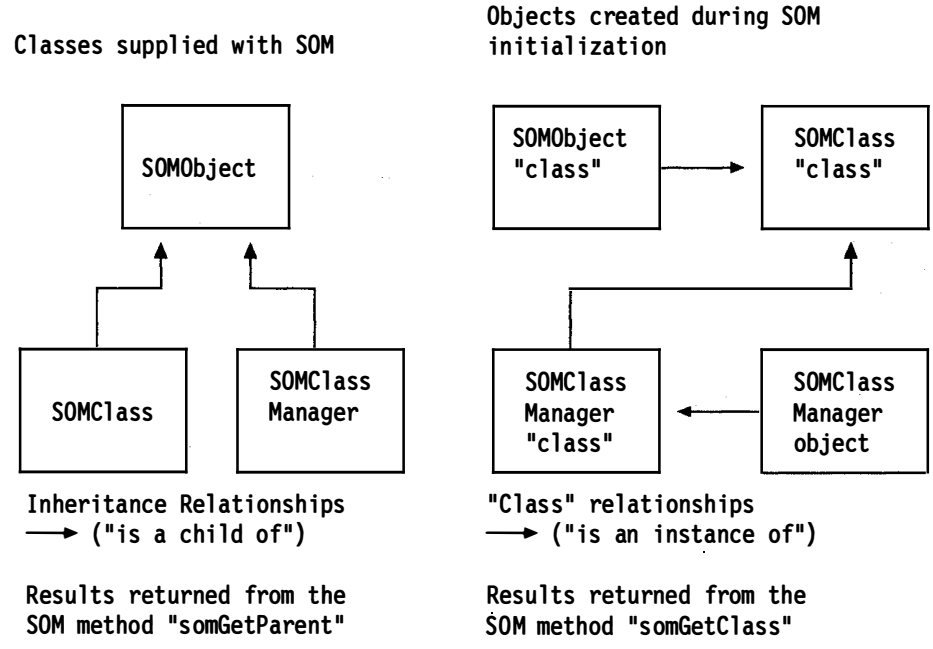


Figure 1-1. Basic SOM Runtime Environment

The classes supplied with SOM are SOMObject, SOMClass, and SOMClassMgr.

SOMObject is the root class for all SOM classes. It defines the essential behavior common to all SOM objects. All user-supplied SOM classes should be derived from this class. No penalty is associated with inheriting from SOMObject, because it has no instance data. What it does contribute, however, is a basic suite of methods that define the common behavior needed in all SOM objects.

The methods that an object responds to are referred to as *instance* methods, because any object instance can perform them. But instance methods cannot be used unless an object instance already exists. To create object instances in the first place, class methods (sometimes called *factory* methods or *constructors*) must be used. Class methods are simply the methods that an object's class responds to.

Because SOM classes are real objects, and all objects are instances of some class, it follows that a SOM class object must also be an instance of some (other) class. We use a special term when talking about the class of a class—namely, *metaclass*.

Just as an object's instance methods are given in its class description (and managed by its run-time class object), the class methods for an object are listed in the description of its metaclass.

Notice that there is a distinct difference between the notions of "parent" class and "metaclass." The parent of a class is another class from which instance methods and state descriptions can be inherited. The "metaclass," on the other hand, provides the factory methods for a class, not instance methods. In general, a class' parent class and its metaclass always will be different classes. Furthermore, a

metaclass has its own inheritance hierarchy (through its parent), which is a completely independent matter.

The distinction between instance methods and class methods, as well as that between objects, classes, and metaclasses, is shown in Figure 1-2.

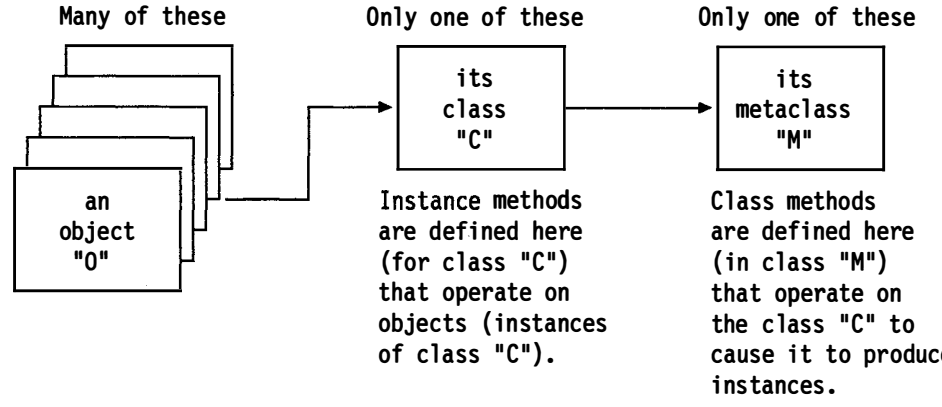


Figure 1-2. Instance Methods versus Class Methods

SOMClass is the root class for all SOM metaclasses. It defines the essential behavior common to all SOM class objects. In particular, it has two generic methods for manufacturing object instances (**somNew** and **somRenew**), and a suite of methods for constructing classes. It also has methods that can be used to dynamically obtain (or augment) information about a class and its methods at run time. Because SOMClass is a subclass of SOMObject, classes inherit the same set of basic instance methods common to all SOM objects. This is what allows SOM classes to be real objects in the SOM run-time environment. SOMClass also has the unique distinction that it is its own metaclass. Look again at Figure 1-1 on page 1-5. The left-hand side shows the parent-child relationships between the built-in SOM classes, and the right-hand side shows the class (and metaclass) relationships.

A single instance of a special SOMClassMgr class is created during SOM initialization, and is referred to as the SOMClassMgrObject. Its job is to maintain a registry of all SOM classes that exist within the current process, and to assist in the dynamic loading and unloading of class libraries packaged as OS/2 Dynamic Link Libraries (DLL). When a class is referenced for the first time, the SOMClassMgrObject will load the appropriate OS/2 DLL and construct a run-time instance of the class.

A user-written class can designate as its metaclass either SOMClass, or another user-written class descended from SOMClass. If a metaclass is not explicitly specified, it defaults to that associated with the class' parent.

The classes that make up the SOM run-time enviroment (SOMObject, SOMClass, and SOMClassMgr) are packaged in a DLL named SOM.DLL. This library also contains a collection of related functions for initializing and customizing the SOM run-time environment. All of the functions and methods contained in SOM.DLL conform to the OS/2 2.0 system linkage conventions.

Chapter 2. The SOM Development Process

Whether you are developing code that you will implement in the form of one or more SOM classes, or writing code that makes use of existing SOM classes, the development process starts with a class definition.

The public definition of a SOM class is given in a formal specification language that is independent of the programming language used in its implementation. The public form of the class definition is kept in a *<class stem>*.SC file. A more complete specification of a class (that can include private interfaces and other implementation detail) would typically be provided in a language-specific form.

The SOM Compiler that IBM provides in the Developer's Toolkit for OS/2 2.0 processes both the language-independent form of a class specification, and an extended form that is specific to the C language. This second form is given in a *<class stem>*.CSC file and can be used by the SOM Compiler to generate the language-independent form (that is, a .SC file). The language-independent form is the public interface that can be used both to access the class and its instances, and to subclass it. The language-specific form, on the other hand, is used only by the class implementer, and is not typically offered for public scrutiny.

As it happens, the C language-specific form of the SOM class specification language is nearly identical to the language-independent form, and so both are described here in "Object Interface Definition Language" on page 2-2.

The SOM Compiler processes class specifications and provides for the production of binding files that bridge the gap between SOM and the object model used in various object-oriented languages, or allow SOM to be used in non-object-oriented languages. In particular, the OS/2 SOM Compiler currently produces a complete set of bindings for the C language. Vendors of other programming languages might also offer SOM bindings. Check with your language vendor about possible SOM support.

The C-language bindings come in the form of macros and functions that are included in your C-language source program and subsequently compiled by the C compiler to produce a running program.

More information about the operation and command syntax of the SOM Compiler is given in "SOM Compiler" on page 2-17.

Object Interface Definition Language

SOM Object Interface Definition Language (OIDL) files provide the basis for generating binding files that enable programming languages to use and provide SOM objects and their definitions (referred to as *classes*). Each OIDL file defines the complete interface to a class of SOM objects.

OIDL files come in different forms for different languages. The different forms enable a class implementer to specify additional language-specific information that allows the SOM OIDL Compiler (or SOM Compiler, for short) to provide support for constructing the class. Each of these different forms shares a common core language that specifies the exact information that a user must know to use a class. One of the facilities of the SOM Compiler is the extraction of the common core part of a class definition. Thus, the class implementer can maintain a language-specific OIDL file for a class, and use the SOM Compiler to produce the language-neutral core definition as needed.

This section describes OIDL with the extensions to support C-language programming.

As indicated above, OIDL files are compiled by the SOM Compiler to produce a set of language-specific or use-specific binding files. The SOM Compiler will produce seven different files for the C language:

- A public header file for programs that *use* a class. Use of a class includes creating instance objects of the class, calling methods on instance objects, and subclassing the class to produce new classes.
- A private header file, which provides usage bindings to any private methods the class might have.
- An implementation header file, which provides macros and other material to support the implementation of the class.
- An implementation template, which provides an outline of the class' implementation that the class provider can then fill out.
- A language-neutral core definition (as discussed above).
- A private language-neutral core file, which contains private parts of the class' interface.
- An OS/2 .DEF file that can be used to package the class in the form of an OS/2 DLL.

OIDL files can contain the following sections. Required sections are designated by (R); optional by (O):

I. Include section (O)

II. Class section (R)

III. Release Order section (O)

IV. Metaclass section (O)

V. Parent Class section (R)

VI. Passthru section (O)

VII. Data section (O)

VIII. Methods section (O)

The SOM Compiler is somewhat flexible about the order of the sections, but the order listed above is highly recommended. The following describes the purpose and form of each of these sections.

The discussion of each section begins with a brief description of its purpose followed by a section-specific syntax diagram. Syntax diagram constants appear in **boldface** (for example, **#include**, **,**, **=**, **;**, **>**, and **"**), while meta symbols appear in the standard font (for example, [...]). User-supplied elements appear in *italics* (for example, *filename*).

Meta symbols have the following meanings:

- Parentheses (( )) indicate grouped parts of the syntax (for example, (**#include** ...)).
- Square brackets ([ ]) indicate optional parts (for example, [, *parent*]).
- An asterisk (*) designates multiples from 0 or more (for example, [, *name*]*).
- A plus sign (+) designates multiples from 1 or more (for example, [*description3*]+).
- A vertical bar (¦) indicates alternatives. Within a set of alternatives, an underscore will indicate the default if one is defined. (for example, **global** ¦ **local**).

Each syntax diagram is followed by a simple example.

Finally, the parts of the section to be supplied by the user are individually described. Each user-input item is labeled "required" or "optional.". If designated "optional," its default will be given following the explanation.

Include Section

This required section contains an include statement that is a directive to the OIDL preprocessor telling the compiler where to find the class interface definition for this class' parent class, the class' metaclass if the class specifies one, and the private interface files for any ancestor class for which this class overrides one or more of its private methods.

Note: The **include** statements must appear in the order shown below. In addition, the ancestor **include** statements must be in inheritance order (from the root down).

The syntax for an **include** statement is:

[**#include** (<*ancestor*> ¦ "*ancestor*")]*
#include (<*parent*> ¦ "*parent*")
[**#include** (<*metaclass*> ¦ "*metaclass*")]

Example:

```
#include "barfile.sc"
#include "foofile.sc"
#include <metafile.sc>
```

where:

<ancestor>

[optional] is the name of the OIDL file containing the private part of an ancestor class' interface needed in the definition of this class. If *ancestor* is enclosed in angle brackets (< >), the search for the file will begin in system-specific locations. If *parent* is enclosed in double quotation marks (""), the search for the file will begin in the local context, then move to the system-specific locations.

<parent>

[required] is the name of the OIDL file containing the parent class of the class for which the **include** statement is provided. If *parent* is enclosed in angle brackets (< >), the search for the file will begin in system-specific locations. If *parent* is enclosed in double quotation marks (""), the search for the file will begin in the local context, then move to the system-specific locations.

<metaclass>

[optional] is the OIDL file containing the metaclass of the class for which the **include** statement is provided. If *metaclass* is enclosed in angle brackets (< >), the search for the file will begin in system-specific locations. If *metaclass* is enclosed in double quotation marks (""), the search for the file will begin in the local context, then move to the system-specific locations. Refer to the note under "Metaclass Section" on page 2-9 for additional information about when a metaclass include may be omitted.

Class Section

This required section introduces the class, giving its name, attributes, and optionally a description of the class as a whole.

The syntax for a **class** statement is:

class: *name*
    [, **file stem** = *stem*]
    [, **external stem** = *stem*]
    [, **function prefix** = *prefix* ¦
     , **external prefix** = *prefix*]
     , **classprefix** = *prefix*]
    [, **major version** = *number*]
    [, **minor version** = *number*]
    [, **global** ¦ **local**];
    [, **classinit** = *function*];
[*description*]

Example:

```
class: Foo,
       local,
       file stem = foofile,
       external prefix = xx_,
       major version = 1,
       minor version = 3;
-- This is the Foo class. It does nothing.
```

where:

<name>
    [required] is the name for this class.

file stem = *stem*
    [optional] is an attribute that specifies how the compiler is to construct file names for various generated files. The constructed file names will all begin with *stem*.

    Note: This value is also used as the class library name in any generated .DEF file.

    This attribute defaults to the name of the file containing the class-interface definition.

external stem = *stem*
    [optional] is an attribute that governs the formation of external names that appear in the text of the generated output files. On some systems, external names are limited in length to a small number of characters. The SOM compiler uses the value *stem* as the basis for generating short external names. A suffix is appended to *stem* for each external name. Because OS/2 external names may be up to 255 characters in length, short external names are generally not used.

    This attribute defaults to the same value given (or defaulted) by **file stem**.

function prefix = *prefix*
    [optional] is an attribute that directs the compiler to construct method function names by prefixing method names with *prefix*. For example, **function prefix** = **xx_** would result in a function name of **xx_foo** for a method called **foo**.

Note: It is an error to specify both **function prefix** and **external prefix** as attributes of the same class.

This attribute has no default.

external prefix = *prefix*

[optional] is an attribute that is similar to **function prefix** except this attribute will also cause each method function to be an external name, whereas method functions usually are local to the module in which they are defined. Making method functions external can be useful in development environments whose debuggers have difficulty dealing with local procedures.

Note: It is an error to specify both **external prefix** and **function prefix** as attributes of the same class.

This attribute has no default.

classprefix = *prefix*

[optional] is similar to the **function prefix** attribute, but applies only to methods that have the **class** attribute. If you use the **class** attribute (for any methods in the current class definition), you are required to supply a **classprefix** value within the **class** statement.

This attribute has no default.

major version = *number*

[optional] is an attribute that specifies the major version number of this class definition. *Number* must be in the range of 0 to (2**31)-1. This will produce bindings that will verify that any code that purports to implement this class has the same major version number, unless *number* is 0, in which case no test is made.

This attribute defaults to **major version = 0**.

minor version = *number*

[optional] is an attribute that specifies the minor version number of this class definition. *Number* must be in the range of 0 to (2**31)-1. This will produce bindings that will verify that any code that purports to implement this class has the same or higher minor version number, unless *number* is 0, in which case no test is made.

This attribute defaults to **minor version = 0**.

global | **local**

[optional] is an attribute that indicates how binding files are to be linked together. **Local** means link to other files in the local context first. **Global** means bypass the local context. For example, **local** will result in C-language statements like:

```
#include "file.h"
```

Global will result in statements like:

```
#include <file.h>
```

This attribute defaults to **global**.

classinit = *function*

[optional] allows you to provide user-written code that participates in the construction of your class. *function* designates a function you supply that is called when your class is created. This function receives one argument—a pointer to your newly created class object. You can use this function to perform any supplemental processing that you need. If you specify this

attribute in your class definition, a template for this function will be provided automatically in the generated .C file.

By default, your class will not have a user-written classinit function.

description

[optional] is a description of the class as a whole and must be in the form of a comment. Several comment styles are supported including:

```
/*
 * line1
 * line2
 */


-- line 1 (starts with 2 consecutive dashes)
-- line 2


// line 1
// line 2
```

Because OIDL files are used to generate language bindings, comments must be strictly associated with particular syntactic elements, so they will appear at the appropriate points in the output files. Therefore, comments must be placed in OIDL files with more care than in some programming languages. If you do not place your comments with strict adherence to the OIDL section syntax, the comments probably will not appear where you expect to see them.

A fourth style for OIDL comments is referred to as *throw-away* comments. They may be placed anywhere in an OIDL file, because they are not intended to appear in any binding files. Throw-away comments start with the character (#) and end at the end of the line. You can use throw-away comments to "comment-out" sections of an OIDL file.

```
#
# This is an example of a throw-away comment
#
```

Note: Comments will appear in files produced by the SOM Compiler in various forms appropriate to the intended use of the file. Sometimes these files are intended as input to a programming-language compiler. Therefore, it is best to avoid using characters in the body of comments that are not generally allowed in most programming-language comments. For example, the C language does not allow "*/" to occur in a comment, so its use is to be avoided.

Refer to "SOM Compiler Command Syntax" on page 2-20 for information about the effect of global attributes on OIDL comments.

Release Order Section

This optional section contains a **release order** statement that forces the compiler to build certain critical data structures with their items arranged in the order specified. This allows the class interface and implementation to be evolved without requiring programs that use this class be recompiled.

Release order applies to all method names and public data items. If the release order of some method or public data item is not specified, it will default to an implementation-specific order based on its occurrence in the OIDL file. The introduction of new public data items or methods might cause the default ordering of other public data items or methods to change; programs using the class would then need to be recompiled. The SOM Compiler has a "pretty print" mode in which it will reproduce a class definition in a nicely formatted manner with **release order** specified for all relevant items. Programmers are advised to use this facility to update the **release order** statement whenever they make significant additions to a class.

The syntax for a **release order** statement is:

release order: *name* [, *name*]* ;

Example:

```
release order: m1, m2, pd1, m3;
```

where:

name [, *name*]*
: [optional] contains all method names introduced by this class (whether public or private), and the names of any public instance variables. It must not contain the names of any inherited methods (even if they are to be overridden), except as noted below. As the class evolves, new names can be added to the end of this list, but once a name is on this list, it must not be reordered or removed. Doing either will require the recompilation of programs that use this class. If a method named on the list is to be moved up in the class hierarchy (for example, to the parent class of this class), its name should remain just as it is on the current list, but it also must be added to the release-order list for the class that will now introduce it.

Metaclass Section

This optional section specifies the class' metaclass, giving its name and, optionally, a description of the reason for the metaclass, or other comments about its role in this class' interface. If a metaclass is specified, its definition must be included in the **include** section. If no metaclass is specified, the metaclass of this class' parent class will be used. Therefore, if you intend the metaclass of this class' parent class to be used, it is generally best not to specify the metaclass.

A class' metaclass can also be implicitly defined through the combined use of the **class** attribute in the **data** section and the **class** attribute in the **method** section. If you use either of these attributes you must bypass (that is, omit) the **metaclass** section altogether. In this case, your implied metaclass will be a subclass of the metaclass of your parent class.

The syntax for a metaclass description statement is:

metaclass: *name*;
[*description*]

Example:

```
metaclass: FooMeta;
/*
 * This is the metaclass for the Foo class.
 * Note that several forms of comment are allowed.
 */
```

where:

name
: [optional] is the name of the metaclass to be used to create the class object for this class.

  This attribute defaults to the parent's metaclass.

description
: [optional] is a description of the metaclass and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

Note: You can add the attributes **major version**, **minor version**, **file stem**, and **global** ¦ **local** to the **metaclass** statement. If you include at least **file stem**, you do not have to include the metaclass definition in the **include** section.

Parent Class Section

This required section specifies the class' parent class, giving its name and, optionally, a description of the role of the parent class in this class' interface.

The syntax for a **parent class** description statement is:

parent [**class**]: *name*;
description

Example:

```
parent: SOMObject;

// Foo will be directly descended from the SOM root class, SOMObject.
// Note, this is yet another style of comment.
```

where:

name
: [required] is the name of the class of which this class is a subclass.

description
: [optional] is a description of the parent class and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

Note: You may add the attributes **major version**, **minor version**, **file stem**, and **global** ¦ **local** to the **parent class** statement.

Passthru Section

This optional section provides blocks of code to be passed by the compiler into various binding files. The contents of the passed lines are essentially ignored by the compiler and can contain anything that needs to be placed near the beginning of a binding file.

Note: Even comments contained in **passthru** lines are processed without modification.

The syntax for the passthru section is:

[passthru: *language.suffix*, [**before** | **after**];
line 1
line 2
...
endpassthru; [*description*]]*

Example:

```
passthru: C.h:
typedef int *foobar;
#define SIZE 89
endpassthru;
-- The two lines above will be placed in the public binding file for all C
-- language users of this class. Note: the next passthru clause has no
-- comment.
passthru: C.ih;
static char *name;
endpassthru;
```

where:

language
: [required] is the programming language whose binding files are to be affected. Currently only the C language is supported. In the future other languages might be supported.

suffix
: [required] specifies which binding file is to be affected. For C, the currently supported binding files are:

.H	The binding file for all users of the class
.PH	The binding file for users with access to private methods of the class
.IH	The binding file for implementers of the class
.C	The program template file used in implementing the class
.SC	The language-neutral core version of the OIDL file
.PSC	Additional language-neutral information about the private methods in the class
.CS2	A "pretty printed" version of the OIDL file

Note: In general, you would not normally place **passthru** lines into a .C file, because a .C file is usually generated only once and then filled in with source code by a programmer. Subsequent updates to a .C file by the SOM Compiler consist only of the addition of new method templates at the end of the file. Place **passthru** lines needed by the class' implementation code in the .IH file instead. Because the .IH file can be

completely regenerated each time the OIDL class definition is compiled, changes to your **passthru** lines will be automatically included.

before ¦ **after**
[optional] is the attribute that indicates whether **passthru** lines should precede or follow any include statements at the beginning of the binding file. This attribute defaults to **before**.

line
[optional] is a line to be placed near the beginning of the binding file exactly as written.

description
[optional] is a description of the purpose of the **passthru** lines.

Data Section

This optional section lists the instance variables for this class. This section is generally present only in the language-specific version of the class interface definition (a .CSC file). However, it must be present in the public form of the class interface definition if the class contains public instance variables. ANSI C syntax is used to describe these variables.

The syntax for the **data** section is:

data:
[*description1*]
[*declaration* [, **private** ¦ , **public** | , **internal**] [, **class**];
[*description2*]]*

Example:

```
data:
-- This is the instance data for the Foo class.
int FooSize, public;
-- FooSize is a public instance variable.  This means that all users of the
-- Foo class will have access to FooSize.
char *fooName;
-- fooName is a private instance variable; therefore, only the implementer of
-- the Foo class will have access to it.
```

where:

description1
[optional] is a description of the instance data as a whole and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

declaration
[optional] is an ANSI C declaration for an instance variable.

private ¦ **public** ¦ **internal**
is the attribute that controls the type of binding (if any) to be exported for the instance data item. **Internal** prevents any binding from being exported. **Private** causes the binding to be part of private-usage binding files, and **public** causes the binding to be placed in public-usage binding files. For C, the exported binding is a macro of the form **get**_*name*(*obj*), where *name* is the name of the instance data item, and *obj* is an object containing the item. The macro returns an expression representing the item, which may be used on the left- or right-hand side of assignment statements.

This attribute defaults to **internal**.

class

is an attribute that designates that the data item belongs to an implicitly defined metaclass. Data items having this attribute do not appear in instances of the class, but instead appear in the class itself. Methods defined in the **methods** section to have the **class** attribute can be created to manipulate **class** items. If you use this attribute your class definition cannot also contain an explicit **metaclass** section.

By default, data items are not considered as **class** data.

description2

is an optional description of the instance variable and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

Methods Section

This optional section lists the methods to be supported by this class. ANSI C function-prototype syntax is used to define the calling sequence to each method.

The syntax for the **methods** section is:

methods:
[*description1*]
[[**group:** *name*;
    [*description2*]]
[*method prototype*
    [, **public** ¦ , **private**]
    [, **method** ¦ , **procedure**]
    [, **class**]
    [, **offset** ¦ , **name lookup**]
    [, **local** ¦ , **external**]
    [, **use** = *name*];
    [*description3*]]*
[**override:** *method name*
    [, **public** ¦ , **private**]
    [, **class**]
    [, **local** ¦ , **external**]
    [, **use** = *name*];
    [*description4*]]*

Example:

```
methods:
-- This section lists all the methods that are to be
-- introduced by the Foo class, as well as all the
-- methods that Foo inherits and wants to override.
void m1 (int parm1, char parm2);
-- m1 is a public method with a fixed calling sequence.
-- It does not return a value.
int m2 (), private;
-- m2 is a private method with no arguments (except
-- for its target object) that returns an int.
long m3 (int parm1), procedure;
-- m3 will be a simple procedure.  It cannot be overridden.
-- It will still have a "somSelf" argument like any other method.
group: g2;
-- The rest of the methods listed in this section are
-- in group "g2".
int m4 (int numargs, ...);
-- m4 is a public method that has one required argument
-- and any number of additional arguments that can be
-- of any types.
override: somInit;
-- This class will override the implementation of
-- somInit that it inherits from one of its ancestor
-- classes.
override: m6, private;
int m5 (int p1, /* first parameter */
        int p2, /* second parameter */
        char *p3 /* last parameter */);
-- Note how comments can be embedded in the method
-- prototype.
```

where:

description1
[optional] is a description of this class' methods as a whole and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

name
[required in each group statement] is the name of this group. Currently, groups serve no purpose except to improve the readability of the object interface definition.

description2
[optional] is a description of this group of methods as a whole and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

<method prototype>
[required in each method specification] is the ANSI C function prototype that defines the calling sequence to this new method with the two exceptions below:

1. Even though the first parameter to each method is the target object, this object is not mentioned in the method prototype.
2. Each parameter can be preceded by one of, IN, INOUT, or OUT, indicating whether it is an input, input/output, or output parameter.

Comments can occur after the separating comma between parameter specifications and after the last parameter, but before the closing parenthesis (see the declaration of *m5* in the example above).

public ¦ **private**
[optional] is an attribute that indicates to the compiler whether or not this method is part of the public interface to this class. There is no real difference between public and private methods, but the compiler will separate the bindings to these two classes of methods so that bindings for public methods can be distributed without distributing bindings for private methods.

This attribute defaults to **public**.

local ¦ **external**
[optional] is an attribute that directs the compiler to declare the method function for this method as an external name or as a local name (directly accessible only to the compilation unit in which it occurs).

When specified, this attribute overrides the class default for method functions (see the descriptions for **function prefix** and **external prefix** in "Class Section" on page 2-5).

This attribute defaults to the class default.

This attribute has not been implemented at this time.

use = *name*
[optional] is an attribute that tells the compiler that the method function for this method is to have the name *name*. This overrides any class prefix specification.

This attribute has no default.

This attribute has not been implemented at this time.

method ¦ procedure

[optional] is an attribute that indicates whether or not this method can be overridden by a subclass.

If **method** is specified, the method will operate on instances of the class and can be overridden by a method in a subclass.

If **procedure** is specified, the method will be a simple procedure. None of the normal method resolution mechanisms will be used in invoking the method procedure; it will be called directly.

This attribute defaults to **method**.

class

is an attribute that specifies whether the method operates on instances or the class itself.

If **class** is specified the method is considered to be associated with an implicit metaclass. In this case the method will operate directly on the class itself and not on its instances. This is a convenient way of defining *constructors* or *factory methods* that can be invoked when no object instances exist. The implicit metaclass is considered to be a subclass of the parent class' metaclass. If you use the **class** attribute you cannot also define an explicit **metaclass** section. In addition, you must also supply a **classprefix** value in the **class** statement (see "Class Section" on page 2-5).

If this attribute is omitted, the method is assumed to operate on object instances rather than the class.

offset ¦ name lookup

[optional] is an attribute that indicates the preferred invocation binding to be used for this method. Generally, methods are invoked on the basis of information provided by the class that introduces the method definition (called *offset resolution*). However, this requires that the class of the method's target object be known at compile time. Sometimes a method definition is introduced by several classes. For such methods, **name lookup** (which invokes the method using the method's name as a key) is a more appropriate choice.

This attribute defaults to **offset**.

description3

[optional] is a description of the method and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

method name

[optional] is the name of a method introduced by an ancestor class that this class will re-implement. Attributes on an override method have the same meaning as previously stated for new methods.

description4

[optional] is a description of the override method or of your reason for overriding it, or both, and must be in the form of a comment (see "Class Section" on page 2-5 for the discussion about comment styles).

SOM Compiler

The SOM Compiler translates the OIDL source definition of a SOM class into a set of bindings appropriate for a particular programming language. The SOM Compiler supplied with the OS/2 2.0 Toolkit produces a complete set of bindings for the C programming language.

The compiler operates in two phases—a precompile phase and an emission phase. In the first phase a precompiler (SPC.EXE) reads and analyzes a user-supplied class definition and produces intermediate output files containing binary class information, comments, and passthru lines). In the second phase, one or more "emitter" programs (EMITC.EXE, EMITH.EXE, EMITIH.EXE, EMITPH.EXE, EMITDEF.EXE, EMITSC.EXE, EMITPSC.EXE, and EMITCSC.EXE) run to produce C-language binding files. Two additional programs (SPP.EXE and SPP2.EXE) serve as preprocessors for the SOM precompiler phase. The sequencing and execution of all of these programs is directed by the SOM Compiler (SC.EXE).

The output from the emitters, plus user-supplied logic for the class' methods, are subsequently compiled by the C compiler and linked by the OS/2 linker to create an executable program. Executable code can be packaged in self-contained .EXE files or placed in a DLL so the class can be used from many programs.

The emitters for C produce the following output files:

<class-stem>.**C**
: A template for a C source program that implements a class. If you are implementing a SOM class in C, this will become your primary source file. It contains stub procedures for all of the methods defined in the *<class-stem>*.CSC file. Once you have supplied your own logic for these methods, subsequent processing by the .C emitter will make only "incremental" updates to your source file, based on changes that you make to the class definition. These incremental updates consist of additional method templates that are appended to the end of your .C file if you have subsequently defined new methods in your class definition.

  The generated .C source file contains an **include** statement for the .IH file described below.

  If you are writing only "client" code that accesses a SOM class someone else has implemented, you do not need to generate a .C file.

<class-stem>.**H**
: This is the public include file for all C-language programs that need to access the SOM class. It contains a tailored set of C macros for accessing the public methods of the class. It also contains a macro for creating new instances of the class, and includes several other header files (based on the class definition), such as the .H files for the parent class and the metaclass, as well as the principal header file for SOM (SOM.H).

<class-stem>.**IH**
: This file is referred to as the *implementation header*. It is included from the *<class-stem>*.C file and contains most of the automatically generated implementation details about your class. The information found here includes:

  - A C struct defining your class' instance data.
  - C macros for accessing your private instance data.
  - C macros for invoking parent methods that your class has overridden.

- A <*className*>GetData macro used by the method stubs in your .C file to set the "somThis" pointer.
- An apply stub for each of the new methods in the class.
- A <*className*>NewClass procedure for constructing your class object at run time.

<*class-stem*>**.PH**
This file contains use macros for any private methods defined in the class (the .IH file will have a **#include** for this file if your class has private methods). This file should be distributed only to users who need knowledge of (and access to) your private methods.

<*class-stem*>**.DEF**
This file can be used by the OS/2 linker to package your class in a DLL. If you want to combine several classes into a single DLL, merge the **exports** statements from each of their .DEF files into a single .DEF file for the entire DLL. When placing multiple classes in one DLL you also must write a simple C procedure named "SOMInitModule" and include it in the export list. This procedure will be called by the SOM Class Manager whenever your DLL is dynamically loaded, and it should call the <*className*>NewClass routine of each class in your DLL. Like all SOM functions and methods, your "SOMInitModule" routine should adhere to OS/2 2.0 system linkage conventions.

<*class-stem*>**.SC**
The language-neutral form of the SOM class definition. It is a proper subset of the .CSC file with all private implementation detail removed. If you wish to publish your class for others to use, this is the form of the OIDL class definition that you should publish. A .SC file also can be used as input by the SOM Compiler, which can use it to generate a .H file for use by programs that are clients of the class.

<*class-stem*>**.PSC**
A supplement to the language-neutral .SC file that contains information about the private methods of a class. The .PSC file contains everything needed for subsequently generating a .PH file.

<*class-stem*>**.CS2**
A "pretty-printed" form of your original .CSC file, with all of the elements arranged in a canonical form. If you want the SOM Compiler to produce a **release order** section for you automatically, you can obtain it from this file. The .CS2 file will have the same content as the original .CSC file except that its **release order** statement will always be complete (even if the **release order** statement in the .CSC file is not), and a consistent comment style will appear throughout.

Of the files discussed here, two of them (the .CSC file and the .C file) should be treated as primary source materials and kept in your development library. The remaining files can be generated from your .CSC file whenever your class is built, or whenever its definition changes.

Running the SOM Compiler

The SOM Compiler is actually a precompiler and a collection of code emitters that take the intermediate output from the precompiler and produce files for a variety of purposes and in several forms including, C header files, a C implementation template, and language-neutral core interface files.

Assume that you have defined the interface for a class called *Example* and that you placed the definition in a file called EXAMPLE.CSC (.CSC is the standard extension for the C-specific form of an OIDL file). Then you might execute the following statement:

```
SC EXAMPLE
```

to compile EXAMPLE.CSC and produce the desired C-language binding files. SC.EXE uses three user-defined environment variables to :i1.SC.EXE control its operation and determine which emitters to execute: *SMINCLUDE*, *SMEMIT*, and *SMTMP*.

The SMINCLUDE Environment Variable

The SOM Compiler uses an environment variable named SMINCLUDE to locate included class definitions. Because every SOM class will have an include for its parent class definition, you will need to set this variable before running the SOM Compiler. It is similar in form to the OS/2 PATH or DPATH environment variables, in that it can consist of one or more directory names separated by a semicolon (;). Directory names can be specified with absolute or relative path names. For example

```
SET SMINCLUDE=.;..\MYSCDIR;C:\TOOLKT20\C\INCLUDE;
```

The SMEMIT Environment Variable

SMEMIT is used to indicate which emitter programs should be executed. Like the SMINCLUDE environment variable it can consist of a list of items separated by semicolons. Each item designates a particular emitter by the name of the file extension the emitter produces. For example

```
SET SMEMIT=H;IH;PH;SC;DEF;
```

indicates that the EMITH.EXE, EMITIH.EXE, EMITPH.EXE, EMITSC.EXE and EMITDEF.EXE programs should be executed to produce .H, .IH, .PH, .SC, and .DEF files, respectively. By default all emitted output files are placed in the same directory as the input file named in the SC command. If the SMEMIT environment variable is not defined, the SOM Compiler will perform a syntax check of your class definition but no output will be produced.

The SMTMP Environment Variable

The SMTMP environment variable specifies the name of a directory that the SOM Compiler uses to hold intermediate output files. For example,

```
SET SMTMP=%TMP%
```

tells SC.EXE to use the same directory for temporary files as given by the setting of the TMP environment variable.

As a general rule, the directory indicated by SMTMP should never coincide with the directory used by the SOM Compiler for its input or the emitted output files. If you do not give a setting for the SMTMP environment variable, SC will use the root directory of the current drive for temporary files.

SOM Compiler Command Syntax

The syntax of the command for running the SOM Compiler (SC.EXE) is:

SC [-*options*] *file*[.CSC]

where *options* can be specified individually, as a string of option characters, or a combination of both. Any option that takes an argument either must be specified individually or appear as the final option in a string of option characters.

The *file* specified with the SC command refers to the file containing the OIDL class definition to be compiled. Typically, this file will have a .CSC or .SC extension, but this is not required. If a file exists with the given name, that file will be used. If no such file exists, a .CSC extension is assumed.

The available options are

-C *n*
set the size of the comment buffer (default: *16384*).

-S *n*
set the total amount of string space for names and passthru lines (default: *32768*).

-V
display version information.

-a *name*[=*value*]
add a global attribute. The currently supported attributes are defined below (at the end of this section).

-d *directory*
specify a directory where all files emitted during the execution of this command should be placed. If the -d option is not used, all emitted output files are placed in the same directory as the input .CSC file.

-h or -?
provide usage information for reference.

-i *filename*
specify the name of the OIDL class definition file. Use this option to override the built-in assumption that the input file will have a .CSC extension. Any *filename* you supply with the -i option is used exactly as you provide it.

-r
check that all release-order entries actually exist (default: FALSE).

-s *string*
substitute *string* in place of the contents of the SMEMIT environment variable for the duration of the current SC command. If you supply a list of values, you must enclose the list with double quotation marks (""). You can use the -s option as a convenient way to override the SMEMIT environment variable. For example:

```
SC -s "h;sc" EXAMPLE
```

is equivalent to the following sequence of commands:

```
SET OLDSMEMIT=%SMEMIT%
SET SMEMIT=h;sc
SC EXAMPLE
SET SMEMIT=%OLDSMEMIT%
```

-w

suppress warning messages (default: FALSE).

The only global attributes currently supported by SC.EXE are:

comment = *comment string*

where *comment string* can be one of the following: "/*", "--", or "//". This indicates that comments marked in the indicated way are to be completely ignored by SC.EXE and not retained for subsequent processing by one of the emitters.

Note: Comments indicated by lines whose first non-white-space character is a # are always ignored by SC.EXE. Also note that comments of any form in passthru lines are passed through.

cstyle = *comment style*

controls the form of emitted comments. *Comment style* must be one of **s**, **c**, or + to cause emitted comments to be in "--", "/* */", or "//" form, respectively. The default form is **s**.

ibmc

causes EMITC, EMITH, EMITIH, EMITPH, and EMITDEF to generate code with pragmas specifically intended for the IBM C Set/2 Compiler. This attribute is specified by default.

ci386

causes EMITC, EMITH, EMITIH, EMITPH, and EMITDEF not to generate code with pragmas specifically intended for the IBM C Set/2 Compiler.

Sample VECTOR.CSC file in OIDL

```
#include <somobj.sc>   # Include the parent class definition

class: Vector,
        file stem = vector,
        external prefix = vect,
        classprefix = mvec,
        major version = 1,
        minor version = 0;

-- Demo class definition that implements a simple vector-of-integers
-- object. Vector elements are numbered from 0.  The size of the
-- vector must be set before using the array (otherwise it raises an
-- error) and cannot be changed once it is set.
-- This is a comment.

parent: SOMObject;

passthru: C.h;
/* something to put in the VECTOR.H file */
endpassthru;

passthru: C.ph;
/* something to put in the VECTOR.PH file */
endpassthru;

passthru: C.ih;
/* something to put in the VECTOR.IH file */
endpassthru;

data:

    integer4 (*body)[], public;

    -- <body> is a public instance variable, and therefore accessible in
    -- any code that has access to this class.  It is accessed via a macro
    -- of the form get_body(obj) where obj is the object whose instance of
    -- body you are accessing.  The macro expression can be used on the
    -- right- or left-hand side of an assignment.

    int size;

    -- <size> is a private instance variable, and therefore accessible
    -- only in the code that participates in the implementation of this
    -- class.

methods:

SOMAny *newVectorOfSize(int size), class;

-- Creates a new instance of Vector and sets its size to <size>.

integer4 get (int i);

-- Returns the <i>th element in the vector.  Raises an error if
-- this vector does not have an <i>th element.

void set (int i, integer4 value);
```

```
-- Set the <i>th element of this vector.  Raises an error if
-- this vector does not have an <i>th element.

void setMany (int start, int num, ...);

-- Set several elements at once.  You must provide <num> elements, and
-- they will be assigned to vector elements starting with <start>.
-- Raises an error if this vector does not include each of the positions
-- covered by the assignment.

void setAll (integer4 value),
        private;

-- Sets all the elements in this vector to <value>.
-- Note: This is a private method and is not available for use unless
-- one has access to the private header file.  It would not appear in
-- the VECTOR.SC file.

void setSize (int size)

-- Set the size of a vector;
-- Use this method to establish the capacity of any Vector instance
-- created by the VectorNew macro (as opposed to instances created
-- with the newVectorOfSize constructor), before putting anything in
-- the vector.

override somInit;
-- Method prototype:
-- void somInit(Vector *somSelf);

-- Add initialization of the vector instance data.

override somDumpSelfInt;
-- Method prototype:
-- void somDumpSelfInt(Vector *somSelf, int level);

-- Add output of the vector instance data.
```

Chapter 3. SOM bindings for C

The form of the SOM Application Programming Interface (API) varies depending on the programming language that you are using. The API described here corresponds to the form that a C-language programmer would use.

Three of the output files produced by the SOM Compiler can be considered as C-language binding files.

<class stem>.**H**
: The public-usage binding file. This file must be included in a C-language program in order to use objects of the class.

<class stem>.**PH**
: The private-usage binding file. This file must be included in a C-language program in order to use the private parts of the class' interface.

  Note: This file includes the *<class stem>*.H file.

<class stem>.**IH**
: The implementation binding file. This file must be included in the source file that implements the class.

  Note: This file includes the *<class stem>*.PH file, if one exists, otherwise it includes the *<class stem>*.H file.

In addition, compilation of a .CSC file for a class can produce or update the *<class stem>*.C file that contains the class' implementation.

All C-language binding files produced by the SOM Compiler are protected with an appropriate **#ifndef** so that they can be included in a C-language compilation unit more than once without harm.

A Brief SOM Programming Primer

Using a Class and Its Objects

Using the SOM C-language bindings for a class and its objects is quite simple. Basically, you need to know only how to declare and initialize object variables and how to invoke their methods. Each of these has a simple form that you can almost always use. The description below illustrates each of these activities beginning with the simple form and followed by more complex forms that are needed in special, less common cases.

Declaring Object Variables

```
<class> *obj;
```

declares *obj* to be pointer to an instance of *<class>*. Because the sizes of SOM objects are not known at compile time, they must be dynamically allocated. Therefore, it is never correct to code a declaration like:

```
<class> obj;  /* never correct */
```

Creating Instances

Instances are generally created by the *<class>*New() macro in a statement like:

```
<class> *obj = <class>New();
```

or

```
<class> *obj;
...
obj = <class>New();
```

Sometimes it is important to pre-allocate space and then convert the space into an object. The *<class>*Renew(obj) macro is provided for this purpose. It might be used in a sequence like:

```
<class> *obj[10];
unsigned char *buffer;
int i;
int size = _somGetInstanceSize(_<class>);

buffer = malloc(10*size);
for (i=0; i<10; i++)
    obj[i] = <class>Renew(buffer+i*size);
```

Such a sequence can be more efficient than many separate memory allocations when a large number of object instances are involved.

Note: _*<class>* is a macro that expands to the expression *<class>***ClassData.classObject**. *<class>***ClassData** is a global structure created for each class. It is initialized when the class is created and contains only one member of which you need to be aware, namely the classObject member. The classObject member is a pointer to the class object for the class.

Invoking Methods

The Normal Case:

The normal way to invoke a method on an object is via a _<*method_name*> macro. For example:

```
_foo(obj, x, y);
```

would invoke the foo macro on an object, *obj*, with parameters, *x*, and *y*. This expression can be used anywhere that a standard function call can be used.

Dealing with Method Name Collisions:

Sometimes you will need to work with classes that independently define methods of the same name. If the providers of these classes anticipated this possibility, and the separate methods have the same signature, then they should have name-lookup bindings. In this case, you can invoke the methods as shown above. However, if the methods do not have the same signature, or they are not provided with name-lookup bindings, you will have to indicate explicitly the class whose method definition you intend to use with a statement like:

```
Bar_foo(obj, x, y);
```

(assuming that *Bar* is the class you intend). You will know that this form is required because the C compiler will give you a warning message about the macro _foo being redefined. After you have changed your source code to fully qualify the references, you can insert:

```
#undef _foo
```

before including the header file (or before each header file if there are several) that provides a second definition of _foo. For example, suppose you were working with two classes, *Bar1*, and *Bar2*, with usage header files BAR1.H, and BAR2.H respectively. Also, suppose each defines a *foo* method (and they are not coordinated as discussed above). Then you might code a file with something like:

```
#include "bar1.h"
#undef _foo
#include "bar2.h"
...
Bar1_foo(obj1, x, y);
...
Bar2_foo(obj2, a, b, c);
```

Invoking Methods by Name:

Occasionally you may need to invoke a method dynamically. For example, you might get the method's name out of a data structure or from user input. To invoke a method dynamically, you need to convert the name into a *somId* and then use one of the methods defined in SOMClass that accepts a method id and returns a pointer to the method procedure for the method.

Note: All of these methods are defined for class objects, so you will need to apply them to your object's class object, not to the object directly. For example, you might code something like:

```
char *name = "foo";
somId mid = SOM_IdFromString(name);
somMethodPtr mp;
SOMAny *obj;

_somFindMethodOk(SOM_GetClass(obj), mid, &mp);
(((void (*)(SOMAny *, int, int))(mp)) (obj, x, y));
```

This example assumes that you know the calling sequence of the method and that it is a static method. Other methods defined in SOMClass can handle other situations. SOMObject defines a set of dispatch methods that also can be used in these situations.

Defining a Class

One of the capabilities of the SOM Compiler is to produce an initial skeleton of your implementation file and then add to this file as you add more methods to your class' interface definition. This makes defining classes very straightforward. Suppose you define a class with this very simple interface:

```
#include <somobj.sc>
class: Simple;
parent: SOMObject;
data:
   int x;
methods:
   int setX(int x);
```

The SOM Compiler will produce an initial implementation file as follows:

```
#define Simple_Class_Source
#include "simple1.ih"

#pragma linkage (system, setX)
SOM_Scope int SOMLINK setX(Simple *somSelf,
                  int x)
{
    SimpleData *somThis = SimpleGetData(somSelf);
    SimpleMethodDebug("Simple","setX");

    return ((int)0);
}
```

All you need to do now is add some contents to setX's definition.

The first line of setX's definition:

```
SimpleData *somThis = SimpleGetData(somSelf);
```

initializes a local variable, *somThis*, to point to a C struct with a data member for each of the instance variables associated with this class, (in this case, *x*). *somThis* is initialized assigning it the result of SimpleGetData(somSelf). SimpleGetData() is a macro defined in SIMPLE.IH that locates Simple's instance data in the larger block of data that contains the complete state of an instance of the Simple class. SimpleGetData can be applied to any object of the Simple class, or any object of a class descended from Simple to locate the Simple instance data in the object.

The second line of setX's definition:

```
SimpleMethodDebug("Simple","setX");
```

is a call to a macro generated for each class. The default definition of this macro produces method-tracing information whenever the global variable **SOM_TraceLevel** is set to a non-zero value. You may, however, assign other definitions of your own devising (for whatever purpose you like) to this macro by placing your definition before the #include "simple.ih" statement. Alternatively, part of the SOM development package is a simple macro (SOM_NoTrace) that you can

assign to macros like SimpleMethodDebug, to completely eliminate the generation of any method-tracing code.

Expansion of the SOM binding macros

The previous discussion of SOM bindings explained and illustrated the way you code in C to use SOM classes and objects. Generally, this is all you will need to know. However, it might be easier for you to debug your programs if you understand the expansion of some of the important SOM macros. This section gives examples and explains the expansion of two important macros.

Method invocation:

```
_foo(obj, x, y);
```

Currently the above statement would expand to one of the following: (assuming *Bar* is the class in which the method *foo* is first introduced)

```
(((somTD_Bar_Foo) somResolve(obj, BarClassData.foo))(obj, x, y));
```

or

```
(((somTD_Bar_Foo)
       _somFindSMethodOk(BarClassData.classObject, somLId_foo))
       (obj, x, y))
```

somTD_Bar_Foo is the type that casts the result of somResolve to an appropriate procedure pointer type so that the call of the *foo* method procedure can be type checked by the C compiler. **somResolve** is a part of the SOM run-time library that performs static method resolution.

Public instance variable access

```
/*
 * Each public instance variable has a macro
 * of the form get_xxxx (where xxxx is the name
 * of the public variable).
 */

currentAcctNumber = get_acctnum (acctobj);
```

The macro shown above illustrates assigning the hypothetical public instance variable *acctnum* (obtained from the object *acctobj*) to a local variable named *currentAcctNumber*. This macro expands to the sequence given below (assuming that the public instance variable is defined in the class *Bar* to be of type ULONG):

```
currentAcctNumber =
    (*((ULONG (*)) ((acctobj)->body+BarClassData.acctnum)))
```

The offset *BarClassData.acctnum* is computed whenever the class *Bar* is instantiated, so that the actual location of the instance variable *acctnum* can change from one compilation to another without any impact on your (client) code.

API for Class Clients

Programs that use or subclass a class are referred to as *client programs*. To be a client of a class, your C-language program must include the <*classstem*>.H file generated by the SOM Compiler. This header contains a customized set of macros for using all of the class' public methods and accessing all of its public instance variables from a C-language program.

A special header (SOM.H) contains, or includes, all of the usage macros for the classes and methods supplied as part of SOM itself. This header is automatically included by any generated <*classstem*>.H file, or can be included explicitly.

<className> NewClass

The C-language interface to a class also includes a function whose name is <*className*>**NewClass**. This is a standard C function (not a method) produced by the SOM Compiler for creating the actual SOM class object. Its interface is given by:

```
SOMAny *<className>NewClass (int majorVersion, int minorVersion);
```

This function will create the class object (if it does not already exist), and return a pointer to it. If the SOM run time has not yet been initialized, this function will initialize it prior to constructing the class. The user-supplied version numbers are checked against the version information built into the class to determine if the class is compatible with the user's expectations. If the class is not compatible, an error condition is raised and (***SOMError**)() is invoked. See the discussion under "somCheckVersion" in the Methods Reference section for more information about version number-checking.

Note: Constructing a class might also cause the dynamic loading or construction of its parent class or classes and its metaclass. After the first call to this routine, subsequent invocations bypass the class construction and simply return a pointer to the existing class object.

<className> New

This macro can be used to create an instance of class <*className*>. It takes no arguments and returns a pointer to a new object instance. If the class object does not already exist, this macro first creates the class object. Its interface is given by:

```
SOMAny *<className>New();
```

If a new instance cannot be created an error condition is raised and (***SOMError**)() is invoked. This macro is a convenient shorthand for applying the built-in somNew method to the appropriate class object, as in

```
SOMAny *aNewInstance;
aNewInstance = _somNew(<className>NewClass());
```

<className> Renew

The <*className*> Renew macro can be used to create an instance of class <*className*> in a block of memory supplied by the calling program. Its operation is identical to the <*className*> New macro, except that no memory is allocated for the new object instance.

```
SOMAny *<className>Renew(SOMAny *buf);
```

The argument *buf* must point to a block of storage large enough to hold an instance of class *<className>* (you can used the built-in SOM method, somGetInstanceSize, to determine the amount of memory required).

_<className>

This macro serves as a convenient way to refer to the class object created by the <className>NewClass macro. You need to be certain that the class has been created prior to any use of this macro.

Method Macros

The C-language macro form of a SOM method invocation consists of an underscore character (_) immediately followed by the method name. This token, when combined with an argument list using standard C function syntax, indicates a method call.

All SOM methods require at least one argument—a pointer to the receiving object, followed by additional arguments, if any, appropriate to the particular method to be invoked.

Note: This first argument (the receiving object) is not listed when the method definition is specified in the class definition, because this is more an artifact of the method resolution mechanism than a logical argument to the method. However, when the method is actually invoked, the receiving object must be passed to the method as the first parameter. This argument designates the object to which the method applies.

If you use a C expression to represent or compute this argument, *you must confine your usage to expressions without side-effects,* as this first argument is actually evaluated twice by the macro expansion. In particular, the following usage is always incorrect:

```
HPS targetPresentationSpace;
SOMClass *VisualObject;
...
_displayYourself (_somNew(VisualObject), targetPresentationSpace);
/* _somNew() is incorrect in this context */
```

Instead, code:

```
SOMAny *temp;
...
temp = _somNew (VisualObject);
_displayYourself (temp, targetPresentationSpace);
```

The correct code above replaces an expression with side-effects (*_somNew()*), with one that can be harmlessly evaluated multiple times (*temp*).

For a typical example of method invocation in C, consider these methods from the sample VECTOR.CSC file:

```
methods:

  SOMAny *newVectorWithSize(int size), class;
  void set (int i, integer4 value);
  integer4 get (int i);
  void setMany (int start, int num, ...);
```

The first three of these could be invoked using the following code sequence:

```
SOMAny aVector;
int i;

VectorNewClass(1,0); /* Ensure that the class exists */
aVector = _newVectorOfSize(_Vector,5);

/* Set aVector to the 1st 5 even numbers */
for (i=0; i<5; i++)
    _set(aVector, i, i*2);

printf("The value of the 3rd element is %ld\n",
       _get(aVector,2)); /* Elements are 0-origin */
```

The _newVectorOfSize method is a constructor for Vector objects (indicated by the *class* attribute in its definition in VECTOR.CSC). Because this method operates on the Vector class rather than on an instance the receiving object is indicated as _Vector. The other methods in this example then operate on the newly created instance, *aVector*.

Methods that are defined to take a varying number of arguments cannot be invoked using macros like those shown above (C macros can use only a fixed number of arguments). Instead, the SOM Compiler generates a special function for each method requiring a variable argument list. These functions have "va_" as a prefix to the method name. The setMany method shown above could be invoked as follows

```
_setSize(aVector,7);
va_setMany(aVector, 2, 5, 4L, 6L, 8L, 10L, 12L);
```

Note: SOM uses the convention that all method names begin with a lowercase letter; uppercase characters are used to emphasize word boundaries within the method name. However, you are free to use any convention you like for methods that you define. Another SOM convention is that all public static methods that have the same (case-sensitive) name, also should have a common signature. That is, they should return values of the same type, and take arguments that agree in number and type. Consequently, it is good practice to use a unique prefix or suffix whenever practical to avoid accidental duplication of method names.

SOM_Resolve and SOM_ResolveNoCheck

In addition to the individual C-language macros for invoking methods, SOM supplies two macros (SOM_Resolve and SOM_ResolveNoCheck) that can be used to invoke any *static* method (static methods are those accessible through offset resolution). The difference between these two macros is that SOM_Resolve performs consistency checking on its arguments while SOM_ResolveNoCheck, which is faster, does not. Both macros require the same three arguments as shown below:

```
SOM_Resolve(<receiver>,<className>,<methodName>)
SOM_ResolveNoCheck(<receiver>,<className>,<methodName>)
```

<receiver> is the object to which the method will apply. As with the method macros, *<receiver>* should be given as an expression without side-effects. *<className>* is the name of the class of the receiving object, and *<methodName>* is the name of the desired method. These names are supplied as simple tokens rather than strings. The macro evaluates to an expression that represents the actual entry-point address of the method procedure. This value can be stored in a variable and retained so that subsequent method resolutions directed

to the same receiving object can be bypassed altogether. Cast this result to be of the same type as the method procedure to which it refers.

Public Instance Variable Macros

Instance variables that have been declared to be public can be accessed directly by clients of a class, using a macro of the form:

```
get_<instanceVariableName>(<targetObject>)
```

Here, <*instanceVariableName*> is the name of the public instance variable to access, and <*targetObject*> is a pointer to the object where the instance variable resides. This macro can be used on either side of the equal sign (=) in a C assignment statement.

API for Class Implementers

Programmers who implement SOM classes in C can use all of the client programmer APIs, as well as several additional macros that are not available in client programs.

The C-language implementation of a SOM class is contained in one or more .C files. If more than one .C file is needed, one (and *only* one) of them is considered the "primary" file and includes the following **#define** statement:

```
#define <className>_Class_Source
```

This is used in the SOM-generated .IH file to determine when to define various functions associated with the class, such as the *<className>* **NewClass** function. As a matter of course, all .C files must include the .IH file. The skeletal .C files generated by the SOM Compiler contain a **#include** statement for the .IH file; any additional .C files you create for the class should be modelled after these.

Implementation Conventions for Methods

The macros that the SOM Compiler produces for use by method implementers presume certain conventions. Because each SOM method always receives at least one argument—a pointer to the object that the method should apply to, this argument is given the name "somSelf."

"somSelf" is defined to be a pointer to an object that is an instance of class *<className>* or an instance of one of its descendent classes. Consequently, the object pointed to by "somSelf" can contain sections with instance data from any number of classes related through inheritance. A local variable named "somThis" is used by each method implementation to access the instance data in its class' section of the object.

The "somSelf" pointer is always passed to the method procedure as an argument; the "somThis" pointer must be set by each method that needs access to instance data. A custom macro, *<className>*GetData, is provided for each class in its .IH file to derive the appropriate value for the "somThis" pointer from the "somSelf" pointer. This *housekeeping* is performed automatically in the method stubs generated by the SOM Compiler, but you need to be aware of this convention. You can use the "somThis" pointer either explicitly to manipulate your instance data, or implicitly by using custom macros for each item of instance data.

<className> GetData Macro

If you are implementing a class that contains instance variables most of the methods in your class will need addressability to the instance data. The SOM Compiler generates a C typedef with the name *<className>*Data for the structure that it uses to hold your instance data. Methods that will access the data then use the *<className>*GetData macro to establish addressability. This macro must be one of the first executable lines of code in each method, and the value it returns should be assigned to a local variable named "somThis." The SOM Compiler automatically generates the code that accomplishes this in each method stub in a .C file.

Here is a sample method implementation from the .C file derived from the sample VECTOR.CSC class definition.

```
#pragma linkage (system, get)                    /* Generated line */
static long SOMLINK get(Vector *somSelf, int i)  /* Generated line */
{                                                /* Generated line */
    VectorData *somThis = VectorGetData(somSelf);/* Generated line */
    VectorMethodDebug("Vector","get");           /* Generated line */

    if (i<_size)                        /* These lines were supplied */
        return (*_body)[i];             /* by the implementer of the */
    else                                /* method.                   */
        SOM_Error (10+SOM_Fatal);       /*                           */

    return ((long) 0);                           /* Generated line */
}                                                /* Generated line */
```

The above example shows a typical usage of the <*className*> GetData macro. It takes a single argument, <*somSelf*>, defined to be a pointer to the object "receiving" the method call, and returns a pointer to the portion of the object that contains the instance data structure for class <*className*>.

Private Instance Variable Macros

If your instance data consists of items with the names <*a, b, c*> the .IH file produced by the SOM Compiler will have custom macros formed by prefixing an underscore character (_) to each item, resulting in macro names of <*_a, _b, _c*>. These macros assume that a local variable named "somThis" has been set to point to the portion of the object instance where your instance data is kept. You can use these macro names for your data items anywhere one of the original names would have been valid.

Parent Method Macros

If you define methods in your class that override those inherited from one of your parent classes, you will frequently want to write your own logic as incremental changes to the parent methods' behavior. That is, at some point within your method, you will need to invoke the parent method to perform its own processing. This could occur at the beginning of your method code, at the very end, or at some point in the middle, depending on the nature of the inherited method and how you intend to supplement or modify its behavior.

To be certain that the method-resolution mechanism will bypass your (overriding) method and invoke the parent method instead, macros are provided in the <*className*>.IH file (for each overridden method) that start with the prefix "parent_". For example, if you have methods <*m1*> and <*m2*> that override parent methods, the macros <*parent_m1*> and <*parent_m2*> can be used in your code to invoke the parent methods when needed.

SOM_ParentResolve

Just as SOM_Resolve is available to clients of a class, the SOM_ParentResolve macro can be used by class implementers, within the body of a method procedure, to obtain the entry-point address of a parent-method procedure. This value can be saved and subsequently used to bypass the method-resolution process when invoking a parent method. Like SOM_Resolve it applies only to static methods. It requires three arguments as shown below:

```
SOM_ParentResolve(<parentClassName>,<className>,<methodName>)
```

<*parentClassName*> is the name of the parent class, <*className*> is the name of the class where the macro appears, and <*methodName*> is the name of the

desired parent method. These names are supplied as simple tokens rather than strings. The macro evaluates to an expression that represents the actual entry-point address of the parent-method procedure. Cast this result to be of the same type as the method procedure to which it refers.

API for General Usage

The interfaces described here can be used both in programs that are are clients of a SOM class and programs that implement SOM classes. To use the general-purpose SOM macros, you need to include <SOM.H>, or any .H file produced from the SOM Compiler.

ID Manipulation

IDs are essentially numbers that uniquely represent strings. They are used in SOM to identify method names, class names, and descriptors. All SOM ID manipulations are case-insensitive, although the original case is always preserved. A set of macros to do convenient ID manipulation is provided as part of the SOM C bindings. The syntax of the ID macros is shown below:

```
#include <som.h> /* or any SOM class .h file */
SOM_CheckId(id)
SOM_CompareIds(id1,id2)
SOM_StringFromId(id)
SOM_IdFromString(str)
```

An ID starts as a pointer to a string (that is, a pointer to a pointer to an array of zero-terminated characters). During its first use with any of the above macros, it is automatically converted to an internal id representation. You can perform this transformation explicitly yourself using the SOM_CheckId macro. Because the representation of an ID changes during this process, a special SOM typedef (**somId**) is provided to declare IDs. You can statically declare an ID, or generate one dynamically from a string. Here is an example of a statically declared ID and a dynamically created one:

```
/* Statically declared ID */

zString example = "exampleMethodName";
somId exampleId = &example;

/* Dynamically created ID */

somId myClassId;
myClassId = SOM_IdFromString ("MyClassName");
```

SOM_CompareIds is a fast and efficient way to determine whether the strings the IDs represent are equal.

There are also a set of functions included in the SOM run-time library that allow you to exert finer control over the creation and use of IDs. These are:

```
int           somRegisterId(somId id);
unsigned long somUniqueKey(somId id);
unsigned long somTotalRegIds(void);
void          somSetExpectedIds(unsigned long numIds);
void          somBeginPersistentIds(void);
void          somEndPersistentIds(void);
```

somRegisterId is identical to the SOM_CheckId macro, but, in addition, returns an indication of whether the string associated with the argument ID was already known (1 indicates that the string was already known; 0 that it has been newly registered).

somUniqueKey returns a numeric value that uniquely represents the string associated with the argument ID.

somTotalRegIds returns the number of IDs that have been registered so far. You can use this value to advise the SOM run time about expected ID usage in later executions of your program by specifying it in a call to **somSetExpectedIds**

somSetExpectedIds, if used, must be called prior to any explicit or implicit invocation of **somEnvironmentNew** (see "SomEnvironmentNew" on page 3-17). It allows you to specify the number of unique IDs you expect to use during the execution of your program. This has the potential of slightly improving your program's space and time utilization (if the value you specify is accurate).

Note: This is the *one* and *only* SOM function that can be invoked prior to **somEnvironmentNew**.

The **somBeginPersistentIds** and **somEndPersistentIds** functions are used to delimit an interval of time for the current thread during which any new IDs that are used (by SOM_CheckId, SOM_IdFromString, or the **somRegisterId** function) are guaranteed to refer only to *static* strings that will not be subsequently freed. (Under normal usage the only time a static string would be freed is when a .DLL in which it resides is unloaded). IDs that are registered within a "persistent ID" interval can be handled quicker and more efficiently because there is no need to create a copy of the strings they reference.

Debugging Facilities

The SOM run time has several facilities for conditionally generating stream-oriented character output. All output characters generated by these facilities ultimately pass through a replaceable procedure called **SOMOutCharRoutine**. The default version of this routine simply writes the character output to *stdout*, but you can replace this procedure with one that routes its output to a window or any other destination of your choice.

Depending on the macros employed, debugging output can be conditionally suppressed or produced based on the setting of three global variables: *SOM_TraceLevel*, *SOM_WarnLevel*, or *SOM_AssertLevel*.

Variable	**Macros/Functions**
SOM_TraceLevel	<*className*>MethodDebug
SOM_WarnLevel	SOM_WarnMsg, SOM_TestC, SOM_Expect
SOM_AssertLevel	SOM_Assert
[**Unconditional**]	somPrintf

<*className*>**MethodDebug**

```
char *class;
char *method;

<className>MethodDebug (class, method);
```

This custom macro is generated as part of the method stubs produced by the .C emitter. It takes two arguments—a class name and a method name—and if SOM_TraceLevel has the value 1 or 2, produces a message each time a method is entered. (Setting SOM_TraceLevel to 2 also causes the methods supplied as part of the SOM run time to generate method trace output.) To suppress the generation of method tracing code, place a line similar to the following in your .C file after the **#include** statement for <*classStem*>.IH:

```
#define <className>MethodDebug(c,m) SOM_NoTrace(c,m)
```

SOM_TestC

```
SOM_TestC (condition);
```

This macro takes an arbitrary Boolean expression as an argument. If the expression evaluates as true (non-zero), execution continues; otherwise **SOM_Error** is invoked with a warning-level error code. If SOM_WarnLevel is set to 1 or 2, a warning message also is produced (the value 2 will include warning messages produced by the SOM run time code).

SOM_WarnMsg

```
char *msg;

SOM_WarnMsg (msg);
```

This macro writes out the string *msg* if SOM_WarnLevel is set to a value of 1 or 2 (a value of 2 also results in warning messages generated from the SOM run-time code).

SOM_Assert

```
integer4 errorCode;

SOM_Assert (condition, errorCode);
```

This macro allows you to place assertions in your code. The assertion is expressed as an arbitrary Boolean expression that is required to evaluate as true (non-zero). If the assertion fails, the SOM_Error macro is invoked using the error code you supply here.

SOM_Expect

```
SOM_Expect (condition);
```

This macro is similar to SOM_Assert, except that if the condition indicated is not true, a SOM_WarnMsg macro is used to produce a warning message to that effect.

somPrintf

```
int charCount;
zString fmt;

charCount = somPrintf (fmt, ...);
```

somPrintf is a function that unconditionally generates character stream output and passes it to (***SOMOutCharRoutine**)().The interface to **somPrintf** is identical to that for the **printf** C library routine.

Validity-Checking Method Calls

In addition to the explicit use of debugging macros, the SOM C-language bindings produce code that automatically performs basic validity checking at run time for all SOM method invocations. If any of these basic checks fail, the SOM_Error macro is used to end the process. Once you have tested your code to your satisfaction and are confident that it is working correctly, you can remove the automatic validity checking by placing the following **#define** in your .C source file prior to the **#include** statement for the <*className*>.IH file.

```
#define SOM_NoTest
```

Error-Handling Facilities

SOM error handling is performed by a user-replaceable procedure that produces a message and an error code and can, if appropriate, end the process where the error occurred. See "Error-Handling Function" on page 4-4 for specifics about the interface to the error-handling procedure.

Each error is associated with a unique integer value referred to as an *error code*. Errors detected by the SOM run-time environment, and their associated error codes, are listed in the Error Codes Appendix.

Errors reported through SOM fall into 3 categories:

SOM_Ignore This classification represents an informational event; it is considered normal, and could be ignored or logged at a user's discretion.

SOM_Warn This category is for unusual conditions that are not considered to be normal events, but that are not severe enough in themselves to require abnormal termination of a program.

SOM_Fatal Errors classified as *fatal* represent conditions that either should not occur or that would result in an intolerable loss of system integrity if processing were allowed to continue. These errors should typically cause the termination of the process in which they occur.

These error classifications are each assigned a single numeric value carried in the low-order digit of the error code.

SOM_Error Macro

```
int errorCode;

SOM_Error (errorCode);
```

The SOM_Error macro takes a SOM error code and invokes the error-handling function passing the error code, the name of the source file, and the line number within the source file where the macro was invoked. The default error-handling function supplied with SOM will provide this information in the form of a message routed through **SOMOutCharRoutine**. Additionally, if the low-order digit of the error code indicates a serious error (value **SOM_Fatal**), the process also is ended; otherwise, control returns to the invoking routine.

SOM_Test

```
SOM_Test (condition);
```

This macro takes an arbitrary Boolean expression as an argument. If the expression evaluates as true (non-zero), execution continues; otherwise, SOM_Error is invoked with a fatal error code.

SOM_GetClass

```
SOMAny *object;
SOMClass *class;

class = SOM_GetClass (object);
```

This macro takes a single argument that is a pointer to an arbitrary SOM object and returns a pointer to its class object. All SOM class objects support a variety of methods that return information about the content and characteristics of the objects

they can create. See "SOMClass" in the Classes Reference section for further information.

SomEnvironmentNew

```
SOMClassMgr *SomEnvironmentNew ();
```

This is a procedure that can be used to initialize the SOM run-time environment explicitly. Although no other SOM facilities [1] can be used until this has been accomplished, it is not generally necessary to call this procedure overtly, because all <*className*>**NewClass** procedures, as well as all <*className*>New and <*className*>Renew macros invoke this function implicitly if it is needed. Upon completion **somEnvironmentNew** returns SOMClassMgrObject. This object is the single instance of SOMClassMgr class that is present in every SOM run-time environment.

1 See "ID Manipulation" on page 3-13 for a minor exception to this rule

Chapter 4. Customization features

SOM is designed to be policy free and highly adaptable. Most of the SOM behavior can be customized by subclassing the built-in classes and overriding methods, but the SOM run time also makes use of a small number of OS/2 facilities. You can override any use of OS/2 system facilities in the SOM run time by supplying your own replacements for the default SOM system-interface routines. You can substitute your replacement routine by placing its entry-point address in one of the following external variables. Because SOM operates within a process, your replacement routine will affect only the active process.

Replacement routines should be written in C and adhere to standard OS/2 system calling conventions.

Replaceable entry points:

SOMCalloc
SOMClassInitFuncName
SOMDeleteModule
SOMError
SOMFree
SOMLoadModule
SOMMalloc
SOMOutCharRoutine
SOMRealloc

Memory-Management Functions

The memory management functions used by the SOM run-time environment are a subset of those supplied in the ANSI C standard library. They have the same calling interface and return the same types of results as their ANSI C equivalents, but include some supplemental error checking. Errors detected in these functions result in the invocation of the (***SOMError**)() function with an appropriate error code.

The correspondence between the SOM memory-management procedure variables and their ANSI standard library equivalents is shown in the following table.

SOM Procedure Variable	ANSI Standard C Library Routine	Return type	Argument types
SOMCalloc	calloc	void *	size_t, size_t
SOMFree	free	void	void *
SOMMalloc	malloc	void *	size_t
SOMRealloc	realloc	void *	void *, size_t

Note: Generally speaking, all of these routines should be replaced as a unit. That is, if you wish to supply your own version of **SOMMalloc** that did all of its allocations as suballocations in a shared memory heap (for instance), you also should supply corresponding **SOMCalloc**, **SOMFree**, and **SOMRealloc** functions that conformed to this same style of memory management.

DLL Management Functions

The SOM run time uses three routines to manage the loading and unloading of DLLs. You can modify the rules that govern the loading and unloading of DLLs by replacing these functions with alternative implementations.

SOMClassInitFuncName

```
zString (*SOMClassInitFuncName) (void);
```

This function returns the name of the function that will initialize all of the classes that are packaged together in a single DLL. The SOM-supplied version of this function returns the string "SOMInitModule." Hence, unless you supply your own alternative to this function, in order to package more than one SOM class in a single OS/2 DLL, you should write a C-language function named "SOMInitModule" that initializes each class in the DLL. For example, if you wanted to create a DLL that had three classes (A, B, and C), your **SOMInitModule** function would look something like this:

```
#include <a.h>
#include <b.h>
#include <c.h>
#pragma linkage (SOMInitModule, system)
void SOMInitModule (integer4 majorVersion, integer4 minorVersion)
    {
    ANewClass (1,0); /* Pass major and minor version numbers */
    BNewClass (2,1); /*  appropriate to each class          */
    CNewClass (majorVersion, minorVersion);
    }
```

Be sure to include **SOMInitModule** as an exported entry point in your .DEF file when creating your DLL.

SOMDeleteModule

```
int (*SOMDeleteModule) (IN somToken modHandle);
```

This function deletes the DLL designated by the value of the parameter, *modHandle*, and returns 0 for success or a non-zero system-specific error code. The parameter, *modHandle*, contains a value returned from **SOMLoadModule** (the DLL loading routine). The default version of this function supplied with SOM returns the OS/2 module handle for the loaded DLL. The default **SOMDeleteModule** routine uses this DLL module handle to invoke the OS/2 **DosFreeModule** function. The result is used as the return value.

SOMLoadModule

```
int (*SOMLoadModule) (IN zString className,
        IN zString fileName,
        IN zString functionName,
        IN integer4 majorVersion,
        IN integer4 minorVersion,
        OUT somToken *modHandle);
```

This function should load the DLL containing the SOM class *className* and return the value 0 for success or a non-zero system-specific error code. The output argument *modHandle* should be used to return a token that can be subsequently used by the **SOMDeleteModule** routine to unload the DLL. The default SOMLoadModule routine supplied with SOM passes back the OS/2 DLL module handle in this argument. The remaining arguments are used as follows:

Argument	Usage
fileName	This argument has the DLL file name, suitable for use with the OS/2 **DosLoadModule** function. It can be either a simple name or a fully qualified name.
functionName	This is the name of the routine to be called if the DLL can be successfully loaded. This routine is responsible for creating the SOM class or classes contained in the DLL. Typically this argument will have the value "SOMInitModule" obtained from **SOMClassInitFuncName** described above. If no "SOMInitModule" entry exists in the DLL, the default version of **SOMLoadModule** looks for a routine with the name <*className*>**NewClass** instead. If neither entry point can be found the **SOMLoadModule** function will fail.
majorVersion	The major version number should be passed to the class initialization function in the DLL.
minorVersion	The minor version number should be passed to the class initialization function in the DLL.

Character Output Function

This routine is invoked from the SOM run-time environment whenever a character is generated by one of the SOM error-handling or debugging macros (see "Error-Handling Facilities" on page 3-16 or "Debugging Facilities" on page 3-14). The default version of this routine supplied with SOM simply writes the character to *stdout* and returns a 1, if successful, or a 0 if not. You might wish to supply a replacement routine to:

- Direct the output to *stderr*
- Record the output in a log file
- Collect characters and handle them in larger chunks
- Send the output to a PM window to display it
- Place the output in the PM clipboard
- Or some combination of these

Use a coding construct similar to the following to install your replacement routine.

```
#include <som.h>
/* Define your replacement routine */
int myReplacementForSOMOutChar (char c)
{
    (Your code goes here)
}
...
/* After the next stmt all output */
/* will be sent to your routine   */
SOMOutCharRoutine = myReplacementForSOMOutChar;
```

Error-Handling Function

SOMError

```
void (*SOMError)(int errorCode, zString fileName, int lineNum);
```

This function inspects the *errorCode* argument and takes appropriate action. The last decimal digit of *errorCode* indicates whether the classification of the error is **SOM_Fatal**, **SOM_Warn**, or **SOM_Ignore** (see "Error-Handling Facilities" on page 3-16 for a discussion of error classifications). In the default **SOMError** fatal errors will end the current process. The remaining two arguments (*fileName* and *lineNum*) indicate the name of the file and the line number within the file where the error was detected.

All error conditions that originate within the classes supplied as part of the SOM run time, leave SOM in an internally consistent state. If you wish to trap them with your error-handling function and resume with some other type of processing, you can do so knowing that all of the methods supplied with SOM behave atomically. That is, they either complete or fail, but if they fail, partial effects are backed out wherever possible so that all SOM methods remain useable and can be re-executed.

Within your own error-handling function you may wish to:

- Record errors in a way appropriate to your application.
- Inform the user through your application's user interface.
- Attempt application level recovery by restarting at a known point.
- Shut down your particular application.

Chapter 5. Classes Reference

SOMObject

Class: SOMObject

Parent: <none>

Metaclass: SOMClass

File stem SOMOBJ

Definition: SOMOBJ.SC

Header file: SOMOBJ.H (always included from SOM.H)

Description: SOMObject is the root class for all SOM classes. It defines the essential behavior common to all SOM objects. As SOMObject has no instance data, it contributes nothing to the size of derived classes.

Notes on subclassing:

All SOM classes are expected to derive from SOMObject. Three methods would typically be overridden by any subclass that has instance data—somInit, somUninit, and somDumpSelfInt. See the descriptions of these methods for further information.

New methods: [Initialization/Termination Group]

somFree
somInit
somUninit

[Access Group]

somGetClass
somGetClassName
somGetSize

[Testing Group]

somIsA
somIsInstanceOf
somRespondsTo

[Dynamic Group]

somDispatchA
somDispatchD
somDispatchL
somDispatchV

[Development Support Group]

somDumpSelf
somDumpSelfInt
somPrintSelf

Inherited methods:

None

Overridden methods:

None

SOMClass

Class: SOMClass

Parent: SOMObject

Metaclass: SOMClass (only class with itself as metaclass)

File stem SOMCLS

Definition: SOMCLS.SC

Header file: SOMCLS.H (always included from SOM.H)

Description: SOMClass is the root class for all SOM metaclasses. It defines the essential behavior common to all SOM classes, In particular, it has two generic methods for manufacturing object instances (somNew and somRenew), and a suite of methods for constructing classes. It also has methods that can be used to dynamically obtain (or augment) information about a class and its methods at run time.

Notes on subclassing:

All SOM classes are expected to have SOMClass or a class derived from SOMClass as their metaclass. Metaclasses define "class" methods (sometimes called "factory" methods or "constructors") that can be used to manufacture objects of any class for which they are the metaclass. If you wish to define your own class methods for your objects, or impart specialized behavior to the generic class methods supplied in SOMClass, you will need to define your own metaclass by subclassing SOMClass or one of its other subclasses. Three methods that SOMClass inherits and overrides from SOMObject would typically be overridden by any metaclass that has instance data - somInit, somUninit, and somDumpSelfInt. See the descriptions of these methods in SOMObject for further information. The new methods introduced in SOMClass that might frequently be overridden are somNew, somRenew, and somClassReady.

Other reasons that you may want to create your own metaclass include tracking object instances, automatic garbage collection, interfacing to a persistent object store, or providing/managing information that is global to a set of object instances.

New methods: [Initialization/Termination Group]

somAddStaticMethod
somClassReady
somInitClass
somOverrideSMethod

[Instance Creation (Factory) Group]

somNew
somRenew

[Access Group]

somGetApplyStub
somGetClassData
somGetClassMtab
somGetInstanceOffset
somGetInstancePartSize
somGetInstanceSize
somGetMethodOffset
somGetName
somGetNumMethods
somGetNumStaticMethods
somGetParent
somGetPClsMtab

somSetClassData

[Testing Group]

somCheckVersion
somDescendedFrom
somSupportsMethod

[Dynamic Group]

somFindMethod
somFindMethodOk

Inherited methods:

somDispatchA
somDispatchD
somDispatchL
somDispatchV
somDumpSelf
somFree
somGetClassName
somGetClass
somGetSize
somIsA
somIsInstanceOf
somPrintSelf
somRespondsTo

Overridden methods:

somDumpSelfInt
somInit
somUninit

SOMClassMgr

Class: SOMClassMgr

Parent: SOMObject

Metaclass: SOMClass

File stem SOMCM

Definition: SOMCM.SC

Header file: SOMCM.H (always included from SOM.H)

Description: One instance of SOMClassMgr is created during SOM initialization. It acts as a run-time registry for all SOM class objects that have been created or dynamically loaded by the current process. Each SOM class automatically registers itself with the SOMClassMgr instance (pointed to by the global variable, SOMClassMgrObject) during the final stage of its initialization.

Notes on subclassing:

You can subclass SOMClassMgr to augment the functionality of its registry (to make it persistent, for example, or to coordinate the name of a class with the location of its code in the file system). If you want your subclass to replace the SOM-supplied SOMClassMgrObject, you can use the somMergeInto method to place the existing registry information from SOMClassMgrObject into your new class-manager object as a final step in the creation of an instance of your subclass. The former SOMClassMgrObject is then freed, and the address of your new class manager is placed in this global variable.

New methods: [Basic Functions Group]

somLoadClassFile
somLocateClassFile
somRegisterClass
somUnloadClassFile
somUnregisterClass

[Access Group]

somGetInitFunction

[Dynamic Group]

somClassFromId
somFindClass
somFindClsInFile
somMergeInto

Inherited methods:

somDispatchA
somDispatchD
somDispatchL
somDispatchV
somDumpSelf
somFree
somGetClass
somGetClassName
somGetSize
somIsA
somIsInstanceOf
somPrintSelf
somRespondsTo

Overridden methods:

somDumpSelfInt
somInit
somUninit

Chapter 6. Methods Reference

somAddStaticMethod

Call Syntax

```
/* Class:  SOMClass
 * Method: somAddStaticMethod
 *
 * Add a static method to a class or
 * override a parent static method
 */
#include <som.h>

SOMClass *receiver;
somId methodId;
zString methodDescriptor;
somMethodProc *method;
somMethodProc *redispatchStub;
somMethodProc *applyStub;

int offset;

offset = _somAddStaticMethod (receiver,
                              methodId,
                              methodDescriptor,
                              method, redispatchStub,
                              applyStub);
```

Uses: Adds or overrides the indicated method to the receiving class. The somAddStaticMethod method is used by the procedure that constructs the SOM class object.

Parameters

receiver The SOM class that should add this (static) method.

methodId A somId that represents the name of this method.

methodDescriptor
A string that describes the types of arguments and the returned value (if any) associated with this method.

method The actual method procedure for this method.

redispatchStub
A procedure with the same calling sequence as this method that re-dispatches the method to one of this class's dispatch functions.

applyStub A procedure that applies a standard data structure for a variable argument list to its target object by calling this method with arguments derived from the data structure. Its calling sequence is the same as that of the dispatch methods defined in SOMObject. This stub supports the dispatch methods used in some classes. In classes where the dispatch functions do not need such a function, this parameter can be null.

Return Value: The offset into the class's static method table for this method is returned. This value can be used subsequently as an index for offset method resolution.

Notes: In general, C-language programmers do not need to use this method, because the SOM Compiler generates all of the code to construct a class in the .IH file for the class.

Related Functions

somInitClass
somOverrideSMethod

Example

```
/* New Method: newMethod1 */

void newMethod1(XXX *somSelf, int arg1);
static char *somMN_newMethod1 = "newMethod1";
static somId somId_newMethod1 = &somMN_newMethod1;
static void  somRD_newMethod1(XXX *somSelf, int arg1)
{
   va_somDispatchV(somSelf, somId_newMethod1,
           somMD_XXX_newMethod1,arg1);
}
static void somAP_newMethod1(XXX *somSelf, somId id,
           char *desc, va_list ap)
{
   int arg1 = va_arg(ap, int);
   _newMethod1(somSelf,arg1);
}

XXXClassData.newMethod1 = (integer2)
  _somAddStaticMethod (XXXClassData.classObject,
        somId_newMethod1,
        somMD_XXX_newMethod1,
        (somMethodProc *) newMethod1,
        (somMethodProc *) somRD_newMethod1,
        (somMethodProc *) somAP_newMethod1);
```

somCheckVersion

Call Syntax

```
/* Class:  SOMClass
 * Method: somCheckVersion
 *
 * Check to see that a class is compatible
 * with the specified version information.
 */
#include <som.h>

SOMClass *receiver;
integer4 majorVersion;
integer4 minorVersion;

int result;

result = _somCheckVersion (receiver,
                           majorVersion,
                           minorVersion);
```

Uses: Check the receiving class for compatibility with the specified major and minor version numbers. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is equal to or greater than *minorVersion*. The major, minor version number pair (0,0) is considered to match any version.

Parameters

receiver The SOM class whose version information should be checked.

majorVersion
This value usually changes only when a significant enhancement or incompatible change is made to a class.

minorVersion
This value changes whenever minor enhancements or fixes are made to a class. Downward compatibility is generally maintained across changes in the *minorVersion* number.

Return Value: Returns 1 (true) if the implementation of this class is compatible with the specified major and minor version number, and false (0) if otherwise.

Notes: This method is usually called immediately after creating the class object to verify that a dynamically loaded class definition is compatible with a using application.

Related Functions

somInitClass

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  myAnimal = AnimalNew();

  if (_somCheckVersion(_Animal, 0, 0))
     somPrintf("Animal IS compatible with 0.0\n");
  else
     somPrintf("Animal IS NOT compatible with 0.0\n");

  if (_somCheckVersion(_Animal, 1, 1))
     somPrintf("Animal IS compatible with 1.1\n");
  else
     somPrintf("Animal IS NOT compatible with 1.1\n");
}
/*
Output from this program:

Animal IS compatible with 0.0
Animal IS NOT compatible with 1.1
*/
```

somClassFromId

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somClassFromId
 *
 * Find a class, given its ID
 */
#include <som.h>

SOMClassMgr *receiver;
somId classId;

SOMClass *class;

class = _somClassFromId (receiver, classId);
```

Uses: Finds the class object, given its ID, if it already exists. Does not load the class.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

classId The ID to use as a key to find the class.

Return Value: Returns NULL if the class object does not yet exist; otherwise, a pointer to the class is returned.

Notes: Use this method instead of somFindClass when you do NOT wish the class to be automatically loaded if it does not already exist in the current process.

Related Functions

somFindClass
somFindClsInFile

Example

```
#include "som.h"

SOMClass *myClass;
char *myClassName = "SampleClass";

myClass = _somClassFromId(SOMClassMgrObject,
                          SOM_IdFromString(myClassName));
if (!myClass)
    somPrintf("Class %s has not yet been loaded.\n", myClassName);
```

somClassReady

Call Syntax

```
/* Class:  SOMClass
 * Method: somClassReady
 *
 * Indicate that a class has been constructed
 * and is ready for normal use.
 */
#include <som.h>

SOMClass *receiver;

_somClassReady (receiver);
```

Uses: This method is invoked when all of the static initialization for the class is finished. The default implementation simply registers the newly constructed class with SOMClassMgrObject. Metaclasses can override this method to augment the class construction sequence in any way that they wish.

Parameters

receiver The class object that has just been constructed.

Return Value: None.

Notes: If you have special processing to do when your class is first created, you can define a metaclass for your class that overrides this method. Typically, the final statement in any overriding method is parent_somClassReady (somSelf) to ensure that your class is properly registered with SOMClassMgrObject.

Never invoke this method yourself; it is invoked automatically at the appropriate time during class construction.

Related Functions

somInitClass

somDescendedFrom

Call Syntax

```
/* Class:  SOMClass
 * Method: somDescendedFrom
 *
 * Test whether the receiving class is derived
 * from the specified class.
 */

#include <som.h>

SOMClass *receiver;
SOMClass *aClassObj;

int result = _somDescendedFrom (receiver, aClassObj);
```

Uses: For programs that use classes as types, this method can be used to ascertain if the type of an object is a subtype of another.

Parameters

receiver The class object to be tested.

aClassObj The potential ancestor class.

Return Value: Returns 1 (true) if the receiving class is a descendent class of the argument class, and 0 (false) if otherwise.

Notes: A class object is considered to be descended from itself for the purpose of this method.

Related Functions

somIsA
somIsInstanceOf

Example

```
#include "animal.h"
#include "dog.h"
/* ------------------------------------------------
   Note: Dog is a subclass of Animal.
   ------------------------------------------------ */
main()
{
  AnimalNewClass(1,1);
  DogNewClass(1,1);

  if (_somDescendedFrom (_Dog, _Animal))
     somPrintf("dog IS descended from animal\n");
  else
     somPrintf("dog is NOT descended from animal\n");
  if (_somDescendedFrom (_Animal, _Dog))
     somPrintf("animal IS descended from dog\n");
  else
     somPrintf("animal is NOT descended from dog\n");
}
/*
Output from this program:
dog IS descended from animal
animal is NOT descended from dog
*/
```

somDispatchX

Call Syntax

```
/* Class:  SOMObject
 * Method: somDispatchX
 *
 * Invoke a method using a dispatch mechanism.
 */
#include <som.h>

SOMAny *receiver;
somId   methodId;
somId   descriptor;

void *ptr;
float8 rnum;
integer4 inum;

ptr  = _somDispatchA (receiver, methodId, descriptor, ...);
rnum = _somDispatchD (receiver, methodId, descriptor, ...);
inum = _somDispatchL (receiver, methodId, descriptor, ...);
       _somDispatchV (receiver, methodId, descriptor, ...);
```

Uses: The somDispatch*X* methods are used to invoke a designated method using a dispatching algorithm appropriate for the class of the specified receiving object. The *actual* object that will receive the designated method is determined by the particular dispatching algorithm associated with the class.

All SOM objects permit the use of a dispatch mechanism to perform method resolution for any method. The default dispatch algorithm supplied in the SOMObject class always selects the specified receiving object as the target of the call and invokes the "apply stub" for the designated method.

Parameters

receiver The object that represents the context in which the dispatch method resolution will occur.

methodId An ID that represents the name of the method to be dispatched.

descriptor An ID that represents a string that describes the arguments (and their types) associated with the target method.

... The remaining arguments are any that are needed for the method to be dispatched.

Return Value: Four families of return values are supported, corresponding to the four forms of the somdispatch*X* method. Within each of the four families, only the largest representation is supported. The four families are:

Pointer This type of result is returned from somDispatchA. It can be cast to be a pointer to a specific type appropriate for the method that is being dispatched.

Floating point
This result is returned from somDispatchD as a float8.

Integer This result is returned from somDispatchL as an integer4.

Void somDispatchV is used for any method that does not return a result.

Notes: These methods make it easier for dynamic domains to bind to the SOM-object protocol boundary. In addition, they determine the appropriate method procedure, then call it with the arguments specified. The default implementations of these methods provided in this class simply look up the apply stub associated with the method and call it. However, other classes can choose to implement any form of lookup they wish. For example, one could provide an implementation of these methods that used the CLOS[1] form of method resolution. For domains that can do so, it generally will be much faster to invoke their methods directly rather than going through a dispatch method. However, all methods can be reached through the dispatch methods.

These methods are declared to take a variable-length argument list, but as with all such methods, SOM requires that the variable part of the argument list be assembled into the standard data structure for variable argument lists before the method is actually invoked. This can be very useful in domains that need to construct the argument list at run time, because they can invoke methods without being able to put the constructed arguments in the normal form for a call. Usually such an operation is impossible in most high-level languages, and the use of assembler language routines would otherwise be necessary.

Related Functions

somGetApplyStub

Example

```
#include "animal.h"
void main()
{
   Animal *myAnimal;
   char *myNoise = "Roar!!!";
   somId somId_setSound;

   myAnimal = AnimalNew();
/* ------------------------------------
   Note: Next two lines are equivalent to
   _setSound(myAnimal, myNoise);
   ------------------------------------ */
   somId_setSound = SOM_IdFromString("setSound");
   _somDispatchV(myAnimal, somId_setSound , (void *) 0, &myNoise);

   _display(myAnimal);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

[1] CLOS is an acronym for Common Lisp Object System

somDumpSelf

Call Syntax

```
/* Class:  SOMObject
 * Method: somDumpSelf
 *
 * Writes a detailed description of the receiving
 * object to (*SOMOutCharRoutine)(char);
 */

#include <som.h>

SOMAny *receiver;
int level;

_somDumpSelf (receiver, level);
```

Uses: Uses **SOMOutCharRoutine** to write a detailed description of this object and its current state. The default implementation produces a header line identifying the receiving object and its class, then invokes somDumpSelfInt to format any instance data in the object.

Parameters

receiver The object to be dumped.

level The nesting level for describing compound objects. It must be greater than or equal to 0. All lines in the description will be preceded by "2 X level" spaces.

Return Value: None.

Notes: This routine writes only the data that concerns the object as a whole, such as class, and uses somDumpSelfInt to describe the object's current state. This approach allows readable descriptions of compound objects to be constructed.

Generally, it is not necessary to override this method, but if it IS overridden it typically will need to be replaced completely.

Related Functions

somDumpSelfInt
somPrintSelf

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  myAnimal = AnimalNew();
  /* ... */
  _somDumpSelf(myAnimal, 0);
}
```

somDumpSelfInt

Call Syntax

```
/* Class:  SOMObject
 * Method: somDumpSelfInt
 *
 * Writes the internal state of the receiving
 * object to (*SOMOutCharRoutine)(char);
 */

#include <som.h>

SOMAny *receiver;
int level;

_somDumpSelfInt (receiver, level);
```

Uses: somDumpSelf invokes this method to write out the instance data stored in the receiving object. The default implementation does nothing (SOMObject has no instance data). Overriding methods can use **SOMOutCharRoutine** to write instance data in a readable format to the SOM output destination.

Parameters

receiver The object to be dumped.

level The nesting level for describing compound objects. It must be greater than or equal to 0. All lines in the description will be preceded by "2 X level" spaces.

Return Value: None.

Notes: Uses (***SOMOutCharRoutine**)() to write out the current state of this object. If your class has instance data, override this method. Begin by calling the parent class form of this method, then write out a description of your class's instance data. This will result in a description of all the object's instance data, from its root ancestor class to its specific class.

This method generally is not invoked directly. Use somDumpSelf instead, which will invoke somDumpSelfInt.

Related Functions

somDumpSelf
somPrintSelf

somFindClass

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somFindClass
 *
 * Finds the class object when given its ID.
 * If the class does not exist, dynamically
 * loads it.
 */
#include <som.h>

SOMClassMgr *receiver;
somId classId;
int majorVersion;
int minorVersion;
SOMClass *class;

class = _somFindClass (receiver, classId, majorVersion, minorVersion);
```

Uses: Returns the class object for the specified class. This may result in dynamic loading. This method first uses somLocateClassFile to obtain the name of the file where the class' code resides, then uses somFindClsInFile.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

classId The ID to use as a key to find the class.

majorVersion The class's major version number.

minorVersion The class's minor version number.

If majorVersion and minorVersion are not both zero, they are used to check the class version information against the caller's expectations.

Return Value: A pointer to the requested class object, or NULL if the class could not be created.

Notes: If the requested class has not yet been created this routine will attempt to load the class dynamically by loading its .DLL and invoking its "new class" procedure.

Related Functions

somFindClsInFile
somLocateClassFile

Example

```
#include "animal.h"

void main()
{
   Animal *bigAnimal;
   SOMClass *myClass;
   char *animalName = "Animal";

   bigAnimal = AnimalNew();
   myClass = _somFindClass (SOMClassMgrObject,
                            SOM_IdFromString(animalName),
                            0, 0);
   somPrintf("myClass: %s\n", _somGetName(myClass));
   _somFree(bigAnimal);
}
/*
Output from this program:
myClass: Animal
*/
```

somFindClsInFile

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somFindClsInFile
 *
 * Same as somFindClass, except the caller provides the
 * file name to be used if dynamic loading is needed.
 *
 */
#include <som.h>

SOMClassMgr *receiver;
somId classId;
int majorVersion;
int minorVersion;
zString file;
SOMClass *class;

class = _somFindClsInFile (receiver, classId, majorVersion, minorVersion, file);
```

Uses: Returns the class object for the specified class. This can result in dynamic loading. This method uses the passed parameter file as the name of the .DLL containing the class.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

classId The ID to use as a key to find the class.

majorVersion The class's major version number.

minorVersion The class's minor version number.

file The name of the .DLL file containing the class.

If majorVersion and minorVersion are not both zero, they are used to check the class version information against the caller's expectations.

Return Value: A pointer to the requested class object, or NULL if the class could not be loaded and created.

Notes: If the requested class has not yet been created this routine will attempt to load the class dynamically by loading its .DLL and invoking its "new class" procedure.

Related Functions

somFindClass

Example

```
#include "som.h"
/*
 *  This program loads a class and creates
 *  an instance of it without having any
 *  external references to it.
 *
 */
void main()
{
   SOMAny *myAnimal;
   SOMClass *animalClass;
   char *animalName = "Animal";
   char *animalFile = "C:\\MYDLLS\\ANIMAL.DLL";

   animalClass = _somFindClsInFile (SOMClassMgrObject,
                                    SOM_IdFromString(animalName),
                                    0, 0,
                                    animalFile);
   myAnimal = _somNew (animalClass);
   somPrintf("The class of myAnimal is %s.\n",
       _somGetClassName(myAnimal));
   _somFree(myAnimal);
}
/*
Output from this program:
The class of myAnimal is Animal.
*/
```

somFindMethod

Call Syntax

```
/* Class:  SOMClass
 * Method: somFindMethod
 *
 * Given a methodId, returns a procedure address
 * and an indication of whether it represents
 * a direct method call or a dispatching function.
 */
#include <som.h>

SOMClass *receiver;
somId methodId;
somMethodProc **m;
int directFlag;

directFlag = _somFindMethod (SOMClass *receiver, methodId, m);
```

Uses: Finds the method procedure associated with *methodId* for the receiving class and sets **m* to it.

Parameters

receiver The class object whose method is desired.

methodId An ID that represents the name of the desired method.

m A pointer to a pointer to a somMethodProc. **m* is set either to NULL (if the method does not exist), or to a value that can be called.

Return Value: Returns 1 (true) when the method procedure can be called directly, and 0 (false) when the method procedure is a dispatch function.

Notes: If the class does not support the specified method, then **m* is set to NULL and the return value is meaningless. Returning a dispatch function does not guarantee that a class supports the specified method; the dispatch might fail.

If a dispatch function is returned (directFlag = = 0) the calling sequence is slightly different than that for a direct method; two additional parameters are required—*methodId* and *descriptorId*.

Related Functions

somFindMethodOk
somSupportsMethod

Example

```
#include "animal.h"
void main()
{
   Animal *myAnimal;
   somId somId_setSound;
   somMethodPtr methodPtr;
   myAnimal = AnimalNew();
/* ---------------------------------------
   Note: Next three lines are equivalent to
       _setSound(myAnimal, "Roar!!!");
   --------------------------------------- */
   somId_setSound = SOM_IdFromString("setSound");
   _somFindMethod (_somGetClass(myAnimal),
                   somId_setSound, &methodPtr);
   methodPtr(myAnimal, "Roar!!!");
/* --------------------------------------- */
   _display(myAnimal);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

somFindMethodOk

Call Syntax

```
/* Class:  SOMClass
 * Method: somFindMethodOk
 *
 * Given a methodId, returns a procedure address
 * and an indication of whether it represents
 * a direct method call or a dispatching function.
 */
#include <som.h>

SOMClass *receiver;
somId methodId;
somMethodProc **m;
int directFlag;

directFlag = _somFindMethodOk (SOMClass *receiver, methodId, m);
```

Uses: Finds the method procedure associated with *methodId* for the receiving class and sets **m* to it. If *methodId* is not supported, an error is raised and execution is halted.

Parameters

receiver The class object whose method is desired.

methodId An ID that represents the name of the desired method.

m A pointer to a pointer to a somMethodProc. **m* is set either to NULL (if the method does not exist), or to a value that can be called.

Return Value: Returns 1 (true) when the method procedure can be called directly, and 0 (false) when the method procedure is a dispatch function.

Notes: If the class does not support the specified method then **m* is set to NULL and the return value is meaningless. Returning a dispatch function does not guarantee that a class supports the specified method; the dispatch might fail.

If a dispatch function is returned (directFlag == 0), the calling sequence is slightly different than that of a direct method; two additional parameters are required—*methodId* and *descriptorId*.

Related Functions

somFindMethod
somSupportsMethod

Example

```
#include "animal.h"
void main()
{
   Animal *myAnimal;
   somId somId_setSound;
   somMethodPtr methodPtr;
   myAnimal = AnimalNew();
/* ---------------------------------------
   Note: Next three lines are equivalent to
       _setSound(myAnimal, "Roar!!!");
   --------------------------------------- */
   somId_setSound = SOM_IdFromString("setSound");
   _somFindMethodOk (_somGetClass(myAnimal),
                     somId_setSound, &methodPtr);
   methodPtr(myAnimal, "Roar!!!");
/* --------------------------------------- */
   _display(myAnimal);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

somFree

Call Syntax

```
/* Class:  SOMObject
 * Method: somFree
 *
 * Release the storage used by the receiver
 * and free the object.
 */
#include <som.h>

SOMAny *receiver;

_somFree (receiver);
```

Uses: Releases the storage associated with the receiving object, assuming that it was created originally by somNew (or another class method that used somNew). No future references should be made to the receiving object. This method calls somUninit before releasing storage.

Parameters

receiver The object to be freed.

Return Value: None.

Notes: This method must be called only on objects that were originally created by somNew, and never on objects created by somRenew. It is not necessary to override this method (override somUninit instead).

Related Functions

somNew
somRenew
somUninit

Example

```
#include "animal.h"

void main()
{
   Animal *myAnimal;
   /*
    * Create an object.
    */
   myAnimal = AnimalNew();

   /* ... */

   /*
    * Free it when finished.
    */
   _somFree(myAnimal);
}
```

somGetApplyStub

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetApplyStub
 *
 * Obtain the apply stub for a given method
 */
#include <som.h>

SOMClass *receiver;
somId methodId;
somMethodProc *applyStub;

applyStub = _somGetApplyStub (receiver, methodId);
```

Uses: This method returns the address of an "apply stub" for the indicated method. An apply stub is a special procedure that accepts the arguments for a particular method in the form of a standard *varargs* data structure. It extracts the arguments, then invokes the method, subsequently returning its result to the original caller. Apply stubs are useful in situations when a static method invocation cannot be constructed at compile time.

Parameters

receiver The class object whose method is desired.

methodId An ID that represents the name of the desired method.

Return Value: Returns the apply stub associated with the specified method. NULL is returned if the method is not supported by this class.

Notes: The calling sequence for any apply stub is shown below.

```
#include <stdarg.h>

somMethodProc *applyStub;
SOMAny *receiver;
somId methodId;
somId descriptor;
va_list arglist;
resultType result;

result = (*applyStub)(receiver, methodId, descriptor, arglist);
```

Related Functions

None.

Example

```
#include "animal.h"
void main()
{
   Animal *myAnimal;
   char *myNoise = "Roar!!!";
   somId somId_setSound;
   somMethodPtr applyStub;

   myAnimal = AnimalNew();
/* ---------------------------------------
   Note: Next three lines are equivalent to
       _setSound(myAnimal, myNoise);
   --------------------------------------- */
   somId_setSound = SOM_IdFromString("setSound");
   applyStub = _somGetApplyStub(
      _somGetClass(myAnimal), somId_setSound);
   applyStub (myAnimal, somId_setSound , (void *) 0, &myNoise);
/* --------------------------------------- */
   _display(myAnimal);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

somGetClass

Call Syntax

```
/* Class:  SOMObject
 * Method: somGetClass
 *
 * Get a pointer to an object's class object
 */
#include <som.h>

SOMAny *receiver;
SOMClass *class;

class = _somGetClass (receiver);
```

Uses: Obtain a pointer to the receiver's class object.

Parameters

receiver is the object whose class is desired.

Return Value: A pointer to the object's class object is returned.

Notes: A slightly faster macro form of this method (SOM_GetClass) also is available. This method is not typically overridden.

Related Functions

somGetClassName

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  int numMethods;
  SOMClass *animalClass;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  numMethods = _somGetNumMethods (animalClass);
  somPrintf ("Number of methods supported by Animal: %d\n", numMethods);
  _somFree (myAnimal);
}
/*
Output from this program:
Number of methods supported by Animal: 24
*/
```

somGetClassData

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetClassData
 *
 * Obtain a pointer to the global ClassData
 * structure associated with the class.
 */
#include <som.h>

SOMClass *receiver;
somClassDataStructure *cds

cds = _somGetClassData (receiver);
```

Uses: This method returns the address of the global ClassData structure associated with the class. Every SOM class has an external data structure named *<className>*ClassData that holds a pointer to the class object and information about the relative offsets of method table entries. This structure can be used by language bindings to assist in the method resolution process.

Parameters

receiver The class object whose ClassData structure is desired.

Return Value: Returns a pointer to the ClassData structure for this class.

Notes: This method is not generally overridden. It is provided for use by the SOM run-time environment.

Related Functions

somSetClassData

somGetClassMtab

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetClassMtab
 *
 * Obtain a pointer to the class' method table.
 */
#include <som.h>

SOMClass *receiver;
somMethodTab *mtab;

mtab = _somGetClassMtab (receiver);
```

Uses: This method returns the address of the class' method table. The method table is a structure containing a pointer to the class object followed by an array of procedure entry addresses, with one entry for each static method defined in this class or any of its parent classes.

Parameters

receiver The class object whose method table is desired.

Return Value: Returns a pointer to the method table of this class.

Notes: This method is not generally overridden.

Related Functions

somGetNumStaticMethods
somGetPClsMtab

Example

```
#include "animal.h"
void main()
{
   Animal *myAnimal;
   somId somId_setSound;
   somMethodPtr methodPtr;
   somMethodTab *methodTable;
   int offset;
   myAnimal = AnimalNew();
/* ---------------------------------------
   Note: Next five lines are equivalent to
       _setSound(myAnimal, "Roar!!!");
   --------------------------------------- */
   somId_setSound = SOM_IdFromString("setSound");
   methodTable = _somGetClassMtab(_somGetClass(myAnimal));
   offset = _somGetMethodOffset(_somGetClass(myAnimal), somId_setSound);
   methodPtr = methodTable->entries[offset];
   methodPtr(myAnimal, "Roar!!!");
/* --------------------------------------- */
   _display(myAnimal);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

somGetClassName

Call Syntax

```
/* Class:  SOMObject
 * Method: somGetClassName
 *
 * Obtain the name of the class of an object.
 */
#include <som.h>

SOMAny *receiver;
zString className;

className = _somGetClassName (receiver);
```

Uses: This method returns the address of a zero-terminated string that gives the name of the class of the receiving object.

Parameters

receiver The object whose class name is desired.

Return Value: Returns a pointer to the name of the class.

Notes: This method is not generally overridden. The address returned is valid until the object's class object is unregistered or freed.

Related Functions

somGetClass

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  SOMClass *animalClass;
  char *className;

  myAnimal = AnimalNew();
  className = _somGetClassName(myAnimal);
  somPrintf("Class name: %s\n", className);
  _somFree(myAnimal);
}
/*
Output from this program:
Class name: Animal
*/
```

somGetInitFunction

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somGetInitFunction
 *
 * Obtain the name of the function that
 * initializes the SOM classes in a DLL.
 */
#include <som.h>

SOMClassMgr *receiver;
zString initFunction;

initFunction = _somGetInitFunction (receiver);
```

Uses: Supplies the name of the initialization function for a DLL containing more than one SOM class.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

Return Value: This method returns the zero-terminated string obtained from (***SOMClassInitFuncName**)(). By default, this exit is set to return the value, "SOMInitModule".

Notes: This method is used by SOMClassMgrObject when it loads a DLL. If the DLL contains an entry point with a matching name, that function is invoked to initialize all of the SOM classes in the DLL. If the DLL does not contain a matching entry point, an entry-point name of the form <*className*>NewClass is invoked. Here <*className*> is the name of the class known to exist in the DLL.

Because of this behavior, if you place only a single SOM class in a DLL, its "NewClass" routine (produced by the SOM Compiler) will be invoked automatically to build the class object when the DLL is loaded. If you package more than one SOM class in a single DLL, you must also supply a C-language function named **SOMInitModule** (or whatever name is ultimately supplied by the somGetInitFunction method) to invoke the class creation routines for all of the classes packaged in the DLL.

Generally speaking, this is not a method that you would ever invoke yourself. But if you were creating a class derived from SOMClassMgr, you might wish to override this method to define your own convention for functions that initialize class DLLs.

Related Functions

somFindClass
somFindClsInFile

somGetInstanceOffset

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetInstanceOffset
 *
 * Obtain the offset of a class' instance data
 * in all of its object instances.
 */
#include <som.h>

SOMClass *receiver;
integer4 offset;

offset = _somGetInstanceOffset (receiver);
```

Uses: This method returns the offset in the body portion of all objects of this class where the class' instance data can be found.

Parameters

receiver The class object whose instance-data offset is desired.

Return Value: Returns the offset (within any object of the receiving class) of the class' instance data.

If a class has no instance data, the value 0 is returned. Use somGetInstancePartSize to determine if any instance data actually exists.

Notes: This method is not generally overridden.

Related Functions

somGetInstancePartSize
somGetInstanceSize

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  SOMClass *animalClass;
  int instanceSize;
  int instanceOffset;
  int instancePartSize;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  instanceSize = _somGetInstanceSize (animalClass);
  instanceOffset = _somGetInstanceOffset (animalClass);
  instancePartSize = _somGetInstancePartSize (animalClass);
  somPrintf ("Instance Size: %d\n", instanceSize);
  somPrintf ("Instance Offset: %d\n", instanceOffset);
  somPrintf ("Instance Part Size: %d\n", instancePartSize);
  _somFree (myAnimal);
}
/*
Output from this program:
Instance Size: 8
Instance Offset: 0
Instance Part Size: 4
*/
```

somGetInstancePartSize

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetInstancePartSize
 *
 * Obtain the size of a class' instance-data
 * section in all of its object instances.
 */
#include <som.h>

SOMClass *receiver;
integer4 size;

size = _somGetInstancePartSize (receiver);
```

Uses: This method returns the amount of space needed in an object of this class to contain the class' instance data.

Parameters

receiver The class object whose instance-data size is desired.

Return Value: Returns the size, in bytes, of the instance data required for this class. This does not include the instance-data space required for this class's ancestor or descendent classes.

If a class has no instance data, 0 is returned.

Notes: This method is not generally overridden.

Related Functions

somGetInstanceOffset
somGetInstanceSize

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  SOMClass *animalClass;
  int instanceSize;
  int instanceOffset;
  int instancePartSize;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  instanceSize = _somGetInstanceSize (animalClass);
  instanceOffset = _somGetInstanceOffset (animalClass);
  instancePartSize = _somGetInstancePartSize (animalClass);
  somPrintf ("Instance Size: %d\n", instanceSize);
  somPrintf ("Instance Offset: %d\n", instanceOffset);
  somPrintf ("Instance Part Size: %d\n", instancePartSize);
  _somFree (myAnimal);
}
/*
Output from this program:
Instance Size: 8
Instance Offset: 0
Instance Part Size: 4
*/
```

somGetInstanceSize

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetInstanceSize
 *
 * Obtain the size of an instance of
 * the receiving class
 */
#include <som.h>

SOMClass *receiver;
integer4 size;

size = _somGetInstanceSize (receiver);
```

Uses: This method returns the total amount of space needed in an object of this class.

Parameters

receiver The class object whose instance size is desired.

Return Value: Returns the size, in bytes, of each instance of this class. This includes the instance-data space required for this class and all of its ancestor classes.

Notes: This method is not generally overridden.

Related Functions

somGetInstanceOffset
somGetInstancePartSize

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  SOMClass *animalClass;
  int instanceSize;
  int instanceOffset;
  int instancePartSize;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  instanceSize = _somGetInstanceSize (animalClass);
  instanceOffset = _somGetInstanceOffset (animalClass);
  instancePartSize = _somGetInstancePartSize (animalClass);
  somPrintf ("Instance Size: %d\n", instanceSize);
  somPrintf ("Instance Offset: %d\n", instanceOffset);
  somPrintf ("Instance Part Size: %d\n", instancePartSize);
  _somFree (myAnimal);
}
/*
Output from this program:
Instance Size: 8
Instance Offset: 0
Instance Part Size: 4
*/
```

somGetMethodOffset

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetMethodOffset
 *
 * Obtain the offset of a static method.
 */
#include <som.h>

SOMClass *receiver;
somId methodId;
int offset;

offset = _somGetMethodOffset (receiver, methodId);
```

Uses: This method returns the specified method's offset in the method-procedure table, assuming this is a static method; returns 0 if it is not. This method is used during the construction of a new class object.

Parameters

receiver The class object whose method table offset is desired.

methodId The ID that designates the name of the method.

Return Value: If the specified method is not a static method, 0 is returned; otherwise the method's offset in the somMethodTab structure is returned.

Notes: The returned value is used to set the offset value in the class data structure of this class. This method is used only when a class formerly defined a method that it now inherits.

This method is not generally overridden.

Related Functions

None.

Example

```
#include "animal.h"
void main()
{
   Animal *myAnimal;
   somId somId_setSound;
   somMethodPtr methodPtr;
   somMethodTab *methodTable;
   int offset;
   myAnimal = AnimalNew();
/* ---------------------------------------
   Note: Next five lines are equivalent to
       _setSound(myAnimal, "Roar!!!");
   --------------------------------------- */
   somId_setSound = SOM_IdFromString("setSound");
   methodTable = _somGetClassMtab(_somGetClass(myAnimal));
   offset = _somGetMethodOffset(_somGetClass(myAnimal), somId_setSound);
   methodPtr = methodTable->entries[offset];
   methodPtr(myAnimal, "Roar!!!");
/* --------------------------------------- */
   _display(myAnimal);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

somGetName

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetName
 *
 * Obtain the name of a class.
 */
 #include <som.h>

 SOMClass *receiver;
 zString className;

 className = _somGetName (receiver);
```

Uses: This method returns the address of a zero-terminated string that gives the name of the receiving class.

Parameters

receiver The class whose name is desired.

Return Value: Returns a pointer to the name of the class.

Notes: This method is not generally overridden. The returned address is valid until the class object is unregistered or freed.

Related Functions

somGetClass
somGetClassName

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  SOMClass *animalClass;
  char *className;

  myAnimal = AnimalNew();
  animalClass = _somGetClass(myAnimal);
  className = _somGetName(animalClass);
  somPrintf("Class Name: %s\n", className);
  _somFree(myAnimal);
}
/*
Output from this program:
Class Name: Animal
*/
```

somGetNumMethods

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetNumMethods
 *
 * Obtain the number of methods
 * available for the receiving class.
 */
#include <som.h>

SOMClass *receiver;
int methodCount;

methodCount = _somGetNumMethods (receiver);
```

Uses: This method returns the number of methods currently supported by this class, including inherited methods (both static and dynamic).

Parameters

receiver The class object whose method count is desired.

Return Value: The total number of methods that are currently available for the receiving class.

Notes: The value returned by this method is the total number of methods currently registered in the receiving class, including static and dynamic methods, whether defined in this class or inherited from a parent class. Because SOM classes are dynamic, other dynamic methods can be added to the class. Furthermore, if the class uses dispatch method resolution, all dynamic methods available in this class might not be formally registered. That is, the dispatch function can permit access to additional methods.

This method is not generally overridden.

Related Functions

somGetNumStaticMethods

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  int numMethods;

  myAnimal = AnimalNew();
  numMethods = _somGetNumMethods(_Animal);
  somPrintf("Number of methods supported by class: %d\n", numMethods);
  _somFree(myAnimal);
}
/*
Output from this program:
Number of methods supported by class: 24
*/
```

somGetNumStaticMethods

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetNumStaticMethods
 *
 * Obtain the number of static methods
 * available for the receiving class.
 */
#include <som.h>

SOMClass *receiver;
int methodCount;

methodCount = _somGetNumStaticMethods (receiver);
```

Uses: This method returns the number of static methods available in this class, including inherited ones.

Parameters

receiver The class object whose static method count is desired.

Return Value: The total number of static methods that are available for the receiving class.

Notes: Static methods are those that can be accessed through the offset resolution mechanism.

This method is not generally overridden.

Related Functions

somGetNumMethods

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  SOMClass *animalClass;
  int nStaticMethods;

  myAnimal = AnimalNew();
  animalClass = _somGetClass(myAnimal);
  nStaticMethods = _somGetNumStaticMethods(animalClass);
  somPrintf("Number of static methods: %d\n", nStaticMethods);
  _somFree(myAnimal);
}
/*
Output from this program:
Number of static methods: 24
*/
```

somGetParent

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetParent
 *
 * Get a pointer to the class' parent class.
 */
#include <som.h>

SOMClass *receiver;
SOMClass *parent;

parent = _somGetParent (receiver);
```

Uses: Obtain a pointer to the receiver's parent class object.

Parameters

receiver The class whose parent class is desired.

Return Value: Returns the parent class of the receiver, if one exists, and NULL otherwise.

Notes: Because all SOM objects inherit from SOMObject, only the class SOMObject will return NULL.

Related Functions

somGetClass

Example

```
/* Note: dog is derived from animal. */
#include "dog.h"
main()
{
  Dog *myDog;
  SOMClass *dogClass;
  SOMClass *parentClass;
  char *parentName;

  myDog = DogNew();
  dogClass = _somGetClass(myDog);
  parentClass = _somGetParent(dogClass);
  parentName = _somGetName(parentClass);
  somPrintf("Name of Parent Class: %s\n", parentName);
  _somFree(myDog);
}
/*
Output from this program:
Name of Parent Class: Animal
*/
```

somGetPClsMtab

Call Syntax

```
/* Class:  SOMClass
 * Method: somGetPClsMtab
 *
 * Obtain a pointer to the parent class' method table
 */
 #include <som.h>

 SOMClass *receiver;
 somMethodTab *mtab;

 mtab = _somGetPClsMtab (receiver);
```

Uses: This method returns the address of the method table for the receiver's parent class.

Parameters

receiver The class object whose parent's method table is desired.

Return Value: Returns a pointer to the method table of the parent class of the receiving class. If this class is a root class (SOMObject), NULL is returned.

Notes: This method is equivalent to:

```
_somGetClassMtab (_somGetParent (receiver))
```

This method is not generally overridden.

Related Functions

somGetClassMtab
somGetNumStaticMethods

Example

```
/* -------------------------------
   Note: Dog is derived from Animal
   ------------------------------- */
#include "dog.h"
#include "animal.h"
void main()
{
   Dog *myDog;
   somId somId_setSound;
   somMethodPtr methodPtr;
   somMethodTab *methodTable;
   int offset;
   myDog = DogNew();
/* ------------------------------------
   Note: Next five lines are equivalent to
       _setSound(myDog, "Woof");
   ------------------------------------ */
   somId_setSound = SOM_IdFromString ("setSound");
   methodTable = _somGetPClsMtab (_somGetClass (myDog));
   offset = _somGetMethodOffset (_somGetParent (_somGetClass(myDog)),
                                 somId_setSound);
   methodPtr = methodTable->entries[offset];
   methodPtr(myDog, "Woof");
/* ------------------------------------ */
   _display(myDog);
   _somFree(myDog);
}
/*
Program Output:
This Animal says
Woof
*/
```

somGetSize

Call Syntax

```
/* Class:  SOMObject
 * Method: somGetSize
 *
 * Obtain the size of an object.
 */
#include <som.h>

SOMAny *receiver;
integer4 size;

size = _somGetSize (receiver);
```

Uses: This method returns the total amount of contiguous space used by the receiving object.

Parameters

receiver The object whose size is desired.

Return Value: Returns the size in bytes of the receiver.

Notes: The value returned reflects only the amount of storage needed to hold the SOM representation of the object. The object might actually be using or managing additional space outside of this area.

This method is not generally overridden.

Related Functions

somGetInstancePartSize
somGetInstanceSize

Example

```
#include "animal.h"
void main()
{
  Animal *myAnimal;
  int animalSize;
  myAnimal = AnimalNew();
  animalSize = _somGetSize(myAnimal);
  somPrintf("Size of animal (in bytes): %d\n", animalSize);
  _somFree(myAnimal);
}
/*
Output from this program:
Size of animal (in bytes): 8
*/
```

somInit

Call Syntax

```
/* Class:  SOMObject
 * Method: somInit
 *
 * Initializes instance data in a newly created object.
 */
#include <som.h>

SOMAny *receiver;

_somInit (receiver);
```

Uses: This method is invoked to cause a newly created object to initialize its instance data.

Parameters

receiver The object to be initialized.

Return Value: None.

Notes: This method initializes instance data in the receiving object. Because instances of SOMObject do not have any instance data, the default implementation does nothing. It is provided to induce consistency among subclasses that require initialization. This method is called automatically as a side effect of object creation.

A companion method (somUninit) is called whenever an object is freed. These two methods should be designed to work together, with somInit priming an object for its first use, and somUninit preparing the object for subsequent release.

In general, if objects of your class contain instance-data items, override the somInit method to set your instance data to an appropriate initial state. When overriding this method, always call the parent-class version of this method *before* doing your own initialization.

Related Functions

somNew
somRenew
somUninit

Example

```
#  animal2.csc:
   include <somobj.sc>
   class: Animal2, local;
   parent: SOMObject;
   data:
     char *sound;
   methods:
      void display();
      override somInit;
      override somUninit;

/* animal2.c: */
   #define Animal2_Class_Source
   #include "animal2.ih"
   #include <string.h>

   SOM_Scope void SOMLINK display (Animal2 *somSelf)
   {
       Animal2Data *somThis = Animal2GetData(somSelf);
   }
   SOM_Scope void SOMLINK somInit (Animal2 *somSelf)
   {
       Animal2Data *somThis = Animal2GetData (somSelf);
       parent_somInit (somSelf);
       _sound = (*SOMMalloc)(100);
       strcpy (_sound, "Unknown Noise");
       somPrintf ("New Animal Initialized\n");
   }
   SOM_Scope void SOMLINK somUninit (Animal2 *somSelf)
   {
       Animal2Data *somThis = Animal2GetData (somSelf);
       (*SOMFree)(_sound);
       somPrintf ("Animal Uninitialized\n");
       parent_somUninit (somSelf);
   }

/* main program */
   #include "animal2.h"
   void main()
   {
      Animal2 *myAnimal;
      myAnimal = Animal2New ();
      _somFree (myAnimal);
   }

/*
Program output:
New Animal Initialized
Animal Uninitialized
*/
```

somInitClass

Call Syntax

```
/* Class:  SOMClass
 * Method: somInitClass
 *
 * Initializes a newly created class object.
 */
#include <som.h>

SOMClass *receiver;
SOMClass *parentClass;
integer4 instanceSize;
int maxStaticMethods;
integer4 majorVersion;
integer4 minorVersion;

_somInitClass (receiver, parentClass, instanceSize,
               maxStaticMethods, majorVersion, minorVersion);
```

Uses: This method is invoked to complete the construction of a newly created class object.

Parameters

receiver The class to be initialized.

cprentCIass The class that the newly created class will inherit from. If NULL is specified, this value defaults to SOMObject. Parent classes must be created prior to the creation of any of their descendant classes.

instanceSize
The amount of space needed to hold the instance data defined for the newly created class. This value should *not* include any space required by parent class(es).

maxStaticMethods
The number of static methods defined for the new class. It should *not* include any static methods defined in the parent classes, even if some of them have been overridden in this class.

majorVersion
The major version number associated with the current implementation of the new class.

minorVersion
The minor version number associated with the current implementation of the new class.

Return Value: None.

Notes: This method is ordinarily invoked from the <*Classname*>NewClass procedure generated by the SOM Compiler from the OIDL class definition. Overriding this method with one of your own is *not* recommended.

Generally speaking, most programmers would never make any direct use of this method. It is provided for use by the C-language bindings produced by the SOM Compiler.

Related Functions

somClassReady
somRegisterClass

Example

```
#include <som.h>
SOMClass *myParentClass;
struct {
   int a, b, c;
} myClassInstanceData;
#define MyClass_MaxMethods 4
#define MyClass_MajorVersion 2
#define MyClass_MinorVersion 1
extern struct MyClassClassDataStructure {
     SOMAny *classObject;
     somMOffset myMethod1;
     somMOffset myMethod2;
     somMOffset myMethod3;
     somMOffset myMethod4;
} MyClassClassData;

/* ... */
_somInitClass (MyClassClassData.classObject,
            "Animal",
            myParentClass,
            sizeof (myClassInstanceData),
            MyClass_MaxMethods,
            MyClass_MajorVersion,
            MyClass_MinorVersion);
```

somIsA

Call Syntax

```
/* Class:  SOMObject
 * Method: somIsA
 *
 * Determine if an object is an instance of a given
 * class (or of one of its descendant classes).
 */
 #include <som.h>

 SOMAny *receiver;
 SOMClass *aClass;
 int result;

 result = _somIsA (receiver, aClass);
```

Uses: Use this method to determine if an object can be treated like an instance of *aClass*.

Parameters

receiver The object to be tested.

aClass The class that the object should be tested against.

Return Value: Returns 1 (true) if the receiving object is an instance of the specified class or of any of its descendant classes, and 0 (false) if otherwise.

Notes: A class object is considered to be descended from itself for the purposes of this method.

Because classes are frequently used as a static typing mechanism, this test is really a way of ascertaining if an object is of a particular static type.

Related Functions

somIsDescendedFrom
somIsInstanceOf

Example

```
#include "animal.h"
#include "dog.h"
/* ------------------------------
   Note: Dog is derived from Animal.
   ------------------------------ */
main()
{
  Animal *myAnimal;
  Dog *myDog;
  SOMClass *animalClass;
  SOMClass *dogClass;

  myAnimal = AnimalNew();
  myDog = DogNew();
  animalClass = _somGetClass (myAnimal);
  dogClass = _somGetClass (myDog);
  if (_somIsA (myDog, animalClass))
     somPrintf ("myDog IS an Animal\n");
  else
     somPrintf ("myDog IS NOT an Animal\n");
  if (_somIsA (myAnimal, dogClass))
     somPrintf ("myAnimal IS a Dog\n");
  else
     somPrintf ("myAnimal IS NOT a Dog\n");
  _somFree (myAnimal);
}
/*
Output from this program:
myDog IS an Animal
myAnimal IS NOT a Dog
*/
```

somIsInstanceOf

Call Syntax

```
/* Class:  SOMObject
 * Method: somIsInstanceOf
 *
 * Determine if an object is an instance of a
 * specific class.
 */
#include <som.h>

SOMAny *receiver;
SOMClass *aClass;
int result;

result = _somIsInstanceOf (receiver, aClass);
```

Uses: Use this method to determine if an object is an instance of a specific class.

Parameters

receiver The object to be tested.

aClass The class that the object should be an instance of.

Return Value: Returns 1 (true) if the receiving object is an instance of the specified class, and 0 (false) if otherwise.

Notes: This method tests an object for inclusion in one specific class. It is equivalent to the expression:

```
(aClass == somGetClass (receiver))
```

Related Functions

somIsDescendedFrom
somIsA

Example

```
#include "animal.h"
#include "dog.h"
/* ------------------------------
   Note: Dog is derived from Animal.
   ------------------------------ */
main()
{
  Animal *myAnimal;
  Dog *myDog;
  SOMClass *animalClass;
  SOMClass *dogClass;

  myAnimal = AnimalNew ();
  myDog = DogNew ();
  animalClass = _somGetClass (myAnimal);
  dogClass = _somGetClass (myDog);
  if (_somIsInstanceOf (myDog, animalClass))
     somPrintf ("myDog is an instance of Animal\n");
  if (_somIsInstanceOf (myDog, dogClass))
     somPrintf ("myDog is an instance of Dog\n");
  if (_somIsInstanceOf (myAnimal, animalClass))
     somPrintf ("myAnimal is an instance of Animal\n");
  if (_somIsInstanceOf (myAnimal, dogClass))
     somPrintf ("myAnimal is an instance of Dog\n");
  _somFree (myAnimal);
  _somFree (myDog);
}
/*
Output from this program:
myDog is an instance of Dog
myAnimal is an instance of Animal
*/
```

somLoadClassFile

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somLoadClassFile
 *
 * Dynamically load a class.
 */
#include <som.h>

SOMClassMgr *receiver;
somId classId;
int majorVersion;
int minorVersion;
zString file;
SOMClass *class;

class = _somLoadClassFile (receiver, classId, majorVersion,
                           minorVersion, file);
```

Uses: The SOMClassMgr object uses this method to load a class dynamically during the processing of somFindClass or somFindClsInFile. A SOM class object representing the class is expected to be created and registered as a result of this action.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

classId The ID representing the name of the class to load.

majorVersion
The major version number used to check the compatibility of the class' implementation with the caller's expectations.

minorVersion
The minor version number used to check the compatibility of the class' implementation with the caller's expectations.

file The name of the DLL file containing the class. The name can be either a simple, unqualified name, without any extension (in which case the OS/2 LIBPATH is searched for a file with the extension, .DLL) or a fully-qualified file name (in which case the OS/2 LIBPATH is not searched).

Return Value: Returns a pointer to the class object, or NULL if the class could not be loaded, or the class object could not be created.

Notes: You can override this method to load or create classes dynamically using your own mechanisms, rather than using the DLL approach supplied with SOM. If you simply wish to change the name of the DLL procedure that is called to initialize the classes in the DLL, override somGetInitFunction instead.

Generally speaking, you will probably never need to make any direct use of this method. Use somFindClass or somFindClsInFile instead.

Related Functions

somFindClass
somFindClsInFile
somGetInitFunction
somUnloadClassFile

somLocateClassFile

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somLocateClassFile
 *
 * Determine the file that holds a
 * class to be dynamically loaded.
 */
#include <som.h>

SOMClassMgr *receiver;
somId classId;
int majorVersion;
int minorVersion;
zString file;

file = _somLocateClassFile (receiver, classId, majorVersion,
                            minorVersion);
```

Uses: The SOMClassMgr object uses this method during somFindclass processing to obtain the name of a file to use when dynamically loading a class.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

classId The ID representing the name of the class to locate.

majorVersion
The major version number used to check the compatibility of the class' implementation with the caller's expectations.

minorVersion
The minor version number used to check the compatibility of the class' implementation with the caller's expectations.

Return Value: Returns the name of the DLL file containing the class. The default implementation in SOMClassMgr simply returns the name of the class.

Notes: If you override this method in a user-supplied subclass the name you return can be either a simple, unqualified name without any extension (in which case the OS/2 LIBPATH is searched for a file with the extension, .DLL) or a fully-qualified file name (in which case the OS/2 LIBPATH is not searched).

Generally speaking, you would not invoke this method directly. It is provided to permit customization of SOMClassMgr through subclassing.

Related Functions

somFindClass
somFindClsInFile
somGetInitFunction
somLoadClassFile
somUnloadClassFile

somMergeInto

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somMergeInto
 *
 * Transfers SOM class registry to
 * another SOMClassMgr instance.
 */
#include <som.h>

SOMClassMgr *receiver;
SOMClassMgr *target;

_somMergeInto (receiver, target);
```

Uses: This method transfers the SOMClassMgr registry information from the receiver to the target. The target object is required to be an instance of SOMClassMgr or one of its subclasses. At the completion of this operation, the target object can function as a replacement for the receiver; the receiver object (which is then in a newly uninitialized state) is freed. If the receiving object is the distinguished instance pointed to from the global variable, SOMClassMgrObject, SOMCLassMgrObject is then reassigned to point to the target object.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

target Another instance of SOMClassMgr or one of its subclasses.

Return Value: None.

Notes: Subclasses that override this method should also transfer their section of the object, then pass this method to their parent as the final step.

Use this method only if you are creating your own special-purpose SOMClassMgr with a derived class. Invoke somMergeInto from your override of the SOMClassMgr somNew method.

Related Functions: None.

Example

```
/*
 * The following example is a hypothetical
 * implementation of an override of the somNew method
 * in a subclass of SOMClassMgr.  It illustrates the
 * proper use of the somMergeInto method.
 */
SOM_Scope SOMAny * SOMLINK somNew (MySOMClassMgr *somSelf)
{
    SOMAny *newInstance;
    static int firstTime = 1;
    /*
     * Permit only one instance of MySOMClassMgr to be created.
     */
    if (!firstTime)
       return (SOMClassMgrObject);
    newInstance = parent_somNew (somSelf);
    /*
     * The next line will transfer the class registry
     * information from SOMClassMgrObject into our
     * new instance.
     */
    _somMergeInto (SOMClassMgrObject, newInstance);
    /* As a result of the above operation
     * SOMClassMgrObject is now set to point to the
     * new instance of MySOMClassMgr.
     */
    firstTime = 0;
    return (newInstance);
}
```

somNew

Call Syntax

```
/* Class:  SOMClass
 * Method: somNew
 *
 * Create a new object instance.
 */
#include <som.h>

SOMClass *receiver;
SOMAny *newObject;

newObject = _somNew (receiver);
```

Uses: This method creates a new instance of the receiving class.

Parameters

receiver The class object that is to create a new instance.

Return Value: A pointer to the newly created object.

Notes: When this method is applied to the class, SOMClass, or to any other metaclass object, it produces a new class object; when applied to a regular class object, it produces an instance of that class.

If the allocation of a new object instance fails, an error condition is raised, and the (***SOMError**)() routine is called.

The C-language bindings produced by the SOM Compiler contain a macro of the form *Classname*New (in the .H file for each class). This is a convenient shorthand for _somNew(_<*Classname*>).

Related Functions

somRenew

Example

```
#include "animal.h"
void main(int argc)
{
   Animal *myAnimal;
/* ------------------------------------------------
   Note: next 2 lines are functionally equivalent to
       myAnimal = AnimalNew();
   ------------------------------------------------ */
   AnimalNewClass(0,0);
   myAnimal = _somNew(_Animal);
   /* ... */
   _somFree(littleAnimal);
}
```

somOverrideSMethod

Call Syntax

```
/* Class:  SOMClass
 * Method: somOverrideSMethod
 *
 * Add a method to a class that
 * overrides a parent method.
 */
#include <som.h>

SOMClass *receiver;
somId methodId;
somMethodProc *method;

_somOverrideSMethod (receiver, methodId, method);
```

Uses: This method can be used instead of somAddStaticMethod when it is known that the class' parent class already supports this method. This method does not require the method descriptor and stub methods that the others do.

Parameters

receiver The class to which the method should be added.

methodId An ID specifying the name of the method to override.

method The procedure entry address for the method.

Return Value: None.

Notes: This method is ordinarily invoked from the <*Classname*>NewClass procedure generated by the SOM Compiler from the Overriding this method with one of your own is *not* recommended.

Generally speaking, you would never use this method directly. It is provided for the use of the C-language binding code produced by the SOM Compiler.

Related Functions

somAddStaticMethod
somInitClass

Example: You will find examples of this method in code generated by the SOM Compiler if you look in the .IH file of any class with overriding methods.

somPrintSelf

Call Syntax

```
/* Class:  SOMObject
 * Method: somPrintSelf
 *
 * Writes a detailed description of the receiving
 * object to (*SOMOutCharRoutine)(char);
 */

#include <som.h>

SOMAny *receiver;
SOMAny *rcvrSelf;

rcvrSelf = _somPrintSelf (receiver);
```

Uses: This method uses **SOMOutCharRoutine** to write a brief string with identifying information about the receiving object. The default implementation gives only the object's class name and its address in memory.

Parameters

receiver The object to be "printed."

Return Value: The receiving object returns a pointer to itself.

Notes: None.

Related Functions

somDumpSelf
somDumpSelfInt

Example

```
#include "animal.h"
main()
{
  Animal *myAnimal;
  myAnimal = AnimalNew ();
  /* ... */
  _somPrintSelf (myAnimal);
  _somFree (myAnimal);
}
/*
Output from this program:
{An instance of class Animal at address 0001CEC0}
*/
```

somRegisterClass

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somRegisterClass
 *
 * Add a class object to the
 * SOM run-time class registry.
 */
#include <som.h>

SOMClassMgr *receiver;
SOMClass *classObj;

_somRegisterClass (receiver, classObj);
```

Uses: This method adds a class object to the SOM run-time class registry maintained by SOMClassMgrObject.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

classObj The class object to add to the SOM class registry.

Return Value: None.

Notes: All SOM run-time class objects should be registered with SOMClassMgrObject. In the C-language bindings created by the SOM Compiler, this is done automatically within the *<Classname>*NewClass procedure generated for each class (during the execution of the somClassReady method).

You never need to invoke this method directly.

Related Functions

somUnregisterClass

somRenew

Call Syntax

```
/* Class:  SOMClass
 * Method: somRenew
 *
 * Create a new object instance
 * using a passed block of storage.
 */
#include <som.h>

SOMClass *receiver;
SOMAny *newObject;

newObject = _somRenew (receiver, newObject);
```

Uses: This method creates a new instance of the receiving class, but uses the space pointed to by *newObject* rather than allocating new space for the object. The default implementation of somRenew automatically re-initializes the object by invoking its somInit method.

Parameters

receiver The class object that is to create a new instance.

newObject A pointer to the space to be used to construct a new object.

Return Value: The pointer supplied by the caller is returned, but is now a pointer to a valid, initialized object.

Notes: No check is made to ensure that the passed pointer addresses enough space to hold an instance of the receiving class. The caller can determine the amount of space necessary by using somGetInstanceSize.

The C-language bindings produced by the SOM Compiler contain a macro of the form <*Classname*>Renew (in the .H file for each class). This is a convenient shorthand for _somRenew(_<*Classname*>).

Related Functions

somGetInstanceSize
somInit
somNew

Example

```
#include "animal.h"
#include <som.h>
#include <stdlib.h>
main()
{
  void *myAnimalCluster;
  Animal *animals[5];
  SOMClass *animalClass;
  int animalSize, i;

  animalClass = AnimalNewClass();
  animalSize = _somGetInstanceSize (animalClass);
  /* Round up to doubleword multiple */
  animalSize = ((animalSize+3)/4)*4;
  /*
   * Next line allocates room for 5 objects
   * in a "cluster" with a single memory-
   * allocation operation.
   */
  myAnimalCluster = malloc (5*animalSize);
  /*
   * The for-loop that follows creates 5 initialized
   * Animal instances within the memory cluster.
   */
  for (i=0; i<5; i++)
     animals[i] = _somRenew (animalClass, myAnimalCluster+(i*animalSize));
  /*
   * Finally, the next line frees all 5 animals
   * with one operation.
   */
  free (myAnimalCluster);
}
```

somRespondsTo

Call Syntax

```
/* Class:  SOMObject
 * Method: somRespondsTo
 *
 * Indicates whether an object
 * supports a given method.
 */
#include <som.h>

SOMAny *receiver;
somId methodId;
int result;

result = _somRespondsTo (receiver, methodId);
```

Uses: Use this method to determine if an object supports a specified method.

Parameters

receiver The object to be tested.

methodId An ID that represents the name of the desired method.

Return Value: Returns 1 (true) if the receiving object supports the specified method, and 0 (false) if otherwise.

Notes: Dynamic typing mechanisms are sometimes based on protocol rather than inheritance. That is, an object is considered to be of the correct type if it can respond to an appropriate set of methods, regardless of its class or the parentage of its class. This method can be used as a basis for a dynamic typing system.

Related Functions

somSupportsMethod

Example

```
/* ---------------------------------------------
   Note: Animal supports a setSound method;
         Animal does not support a doTrick method.
   --------------------------------------------- */
#include "animal.h"
main()
{
  Animal *myAnimal;
  char *methodName1 = "setSound";
  char *methodName2 = "doTrick";

  myAnimal = AnimalNew();
  if (_somRespondsTo(myAnimal, SOM_IdFromString(methodName1)))
     somPrintf("myAnimal responds to %s\n", methodName1);
  if (_somRespondsTo(myAnimal, SOM_IdFromString(methodName2)))
     somPrintf("myAnimal responds to %s\n", methodName2);
  _somFree(myAnimal);
}
/*
Output from this program:
myAnimal responds to setSound
*/
```

somSetClassData

Call Syntax

```
/* Class:  SOMClass
 * Method: somSetClassData
 *
 * Sets the class' pointer to its global
 * ClassData structure.
 */
#include <som.h>

SOMClass *receiver;
somClassDataStructure XyzClassData;

_somSetClassData (receiver, &XyzClassData);
```

Uses: This method sets the class' pointer to its global ClassData structure. Every SOM class has an external data structure named <*className*>ClassData that holds a pointer to the class object and information about the relative offsets of method table entries.

Parameters

receiver The class object whose ClassData structure pointer is to be set.

Return Value: None.

Notes: This method is not generally overridden. It is provided for use by the SOM run-time environment.

Related Functions

somGetClassData

somSupportsMethod

Call Syntax

```
/* Class:  SOMClass
 * Method: somSupportsMethod
 *
 * Indicates whether instances of this
 * class support a given method.
 */
#include <som.h>

SOMClass *receiver;
somId methodId;
int result;

result = _somSupportsMethod (receiver, methodId);
```

Uses: Use this method to determine if instances of the receiving class support a specified method.

Parameters

receiver The class object to be tested.

methodId An ID that represents the name of the desired method.

Return Value: Returns 1 (true) if instances of the receiving class object support the specified method, and 0 (false) if otherwise.

Notes: None.

Related Functions

somRespondsTo

Example

```
/* ---------------------------------------------
   Note: animal supports a setSound method;
         animal does not support a doTrick method.
   --------------------------------------------- */
#include "animal.h"
main()
{
  Animal *myAnimal;
  SOMClass *animalClass;
  char *methodName1 = "setSound";
  char *methodName2 = "doTrick";

  myAnimal = AnimalNew();
  animalClass = _somGetClass(myAnimal);
  if (_somSupportsMethod(animalClass, SOM_IdFromString(methodName1)))
     somPrintf("Animals respond to %s\n", methodName1);
  if (_somSupportsMethod(animalClass, SOM_IdFromString(methodName2)))
     somPrintf("Animals respond to %s\n", methodName2);
  _somFree(myAnimal);
}
/*
Output from this program:
Animals respond to setSound
*/
```

somUninit

Call Syntax

```
/* Class:  SOMObject
 * Method: somUninit
 *
 * Un-initializes the receiving object.
 */
#include <som.h>

SOMAny *receiver;

_somUninit (receiver);
```

Uses: This method performs the inverse of object initialization on the receiving object. It must release any resources acquired by the object during its somInit processing.

Parameters

receiver The object to be un-initialized.

Return Value: None.

Notes: Use this method to clean up anything necessary, such as dynamically allocated storage. However this method does not release the actual storage assigned to the object instance. This method is provided as a complement to somFree, which also releases the storage associated with a dynamically allocated object. Usually you would invoke only somFree (which always calls somUninit). However, in cases where somRenew was used to create an object instance, somFree cannot be called, and you must call somUninit explicitly.

When overriding this method, always call the parent-class version of this method *after* doing your own un-initialization.

Related Functions

somInit
somNew
somRenew

Example

```
#  animal2.csc:
   include <somobj.sc>
   class: Animal2, local;
   parent: SOMObject;
   data:
     char *sound;
   methods:
      void display();
      override somInit;
      override somUninit;

/* animal2.c: */
   #define Animal2_Class_Source
   #include "animal2.ih"
   #include <string.h>

   SOM_Scope void SOMLINK display (Animal2 *somSelf)
   {
       Animal2Data *somThis = Animal2GetData(somSelf);
   }
   SOM_Scope void SOMLINK somInit (Animal2 *somSelf)
   {
       Animal2Data *somThis = Animal2GetData (somSelf);
       parent_somInit (somSelf);
       _sound = (*SOMMalloc)(100);
       strcpy (_sound, "Unknown Noise");
       somPrintf ("New Animal Initialized\n");
   }
   SOM_Scope void SOMLINK somUninit (Animal2 *somSelf)
   {
       Animal2Data *somThis = Animal2GetData (somSelf);
       (*SOMFree)(_sound);
       somPrintf ("Animal Uninitialized\n");
       parent_somUninit (somSelf);
   }

/* main program */
   #include "animal2.h"
   void main()
   {
      Animal2 *myAnimal;
      myAnimal = Animal2New ();
      _somFree (myAnimal);
   }

/*
Program output:
New Animal Initialized
Animal Uninitialized
*/
```

somUnloadClassFile

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somUnloadClassFile
 *
 * Unloads a dynamically loaded
 * class and frees the class object.
 */
#include <som.h>

SOMClassMgr *receiver;
SOMClass *class;
int errorCode;

errorCode = _somUnloadClassFile (receiver, class);
```

Uses: The SOMClassMgr object uses this method to unload a dynamically loaded class during somUnregisterClass processing.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

class The class to unload.

Return Value: Returns 0 if the class was successfully unloaded, or a system-specific non-zero error code from **DosFreeModule**.

Notes: The class object is freed whether or not the class' DLL could be unloaded. If the class was not registered, an error condition is raised and (***SOMError**)() is invoked. This method is provided to permit user-created subclasses of SOMClassMgr to handle the unloading of classes.

Do not invoke this method directly; invoke somUnregisterClass instead.

Related Functions

somLoadClassFile
somRegisterClass
somUnregisterClass

somUnregisterClass

Call Syntax

```
/* Class:  SOMClassMgr
 * Method: somUnregisterClass
 *
 * Removes a class object from
 * the SOM run-time class registry.
 */
#include <som.h>

SOMClassMgr *receiver;
SOMClass *class;
int errorCode;

errorCode = _somUnregisterClass (receiver, class);
```

Uses: This method unregisters a SOM class, unloads its DLL (if it was dynamically loaded), and frees the class object.

Parameters

receiver Usually SOMClassMgrObject (or an instance of a user-supplied subclass of SOMClassMgr).

class The class to unregister.

Return Value: This method returns 0 for a successful completion, or non-zero for a failure.

Notes: This method uses somUnloadClassfile to delete a dynamically loaded class.

Related Functions

somLoadClassFile
somRegisterClass
somUnloadClassFile

Example

```
#include <som.h>

int rc;
/*
 * Assume variable "class" points to
 * a class object to be unregistered
 */
SOMClass *class;

rc = _somUnregisterClass (SOMClassMgrObject, class);
if (rc)
   somPrintf ("Could not unregister %s, error code was: %d.\n,
      _somGetName (class), rc);
```

Appendix A. Error Codes

Value

Symbolic Name and Description

20011

SOMERROR_CCNullClass
Explanation: The somDescendedFrom method was passed a null class argument.

20029

SOMERROR_SompntOverflow
Explanation: The internal buffer used in **somPrintf** overflowed.

20039

SOMERROR_MethodNotFound
Explanation: somFindMethodOk failed to find the indicated method.

20049

SOMERROR_StaticMethodTableOverflow
Explanation: A Method-table overflow occurred in somAddStaticMethod.

20059

SOMERROR_DefaultMethod
Explanation: The somDefaultMethod was called; a defined method probably was not added before it was invoked.

20069

SOMERROR_MissingMethod
Explanation: The specified method was not defined on the target object.

20079

SOMERROR_BadVersion
Explanation: An attempt to load, create, or use a version of a class-object implementation is incompatible with the using program.

20089

SOMERROR_NullId
SOM_CheckId was given a null ID to check

20099

SOMERROR_OutOfMemory
Explanation: Memory is exhausted.

20109

SOMERROR_TestObjectFailure
Explanation: somObjectTest found problems with the object it was testing.

20119

SOMERROR_FailedTest
Explanation: somTest detected a failure; generated only by test code.

20121

SOMERROR_ClassNotFound
Explanation: somFindClass could not find the requested class.

20131

SOMERROR_OldMethod
Explanation: An old-style method name was used; change to an appropriate name.

20149

SOMERROR_CouldNotStartup
Explanation: somEnvironmentNew failed to complete.

20159

SOMERROR_NotRegistered
Explanation: somUnloadClassFile argument was not a registered class.

20169

SOMERROR_BadOverride
Explanation: somOverrideSMethod was invoked for a method that was not defined in a parent class.

20179

SOMERROR_NotImplementedYet
Explanation: The method raising the error message is not implemented yet.

20189

SOMERROR_MustOverride
Explanation: The method raising the error message should have been overridden.

20199

SOMERROR_BadArgument
Explanation: An argument to a core SOM method failed a validity test.

20219

SOMERROR_NoParentClass
Explanation: During the creation of a class object, the parent class could not be found.

20229

SOMERROR_NoMetaClass
Explanation: During the creation of a class object, the metaclass object could not be found.

Glossary

class. A way of categorizing objects based on their behavior and shape. A class is, in effect, a definition of a generic object. In SOM, a class is a special kind of object that can manufacture other objects that all have a common shape and exhibit similar behavior (more precisely, all of the objects manufactured by a class have the same memory layout and share a common set of methods). New classes can be defined in terms of existing classes through a technique known as *inheritance*.

class method. (Also known as *factory method* or *constructor*). A class method of class <X> is a method provided by the metaclass of class <X>. Class methods are executed without requiring any instances of class <X> to exist, and are frequently used to create instances.

constructor. See *class method*.

dynamic method. A method for which offset resolution is not available.

factory method. See *class method*.

ID. A pointer to a unique value that represents a string.

inheritance. The technique of specifying the shape and behavior of one class (called a *subclass*) as incremental differences from another class (called the *parent class* or *superclass*). The subclass inherits the superclass' state representation and methods, and can provide additional data elements and methods. The subclass also can provide new functions with the same method names used by the superclass. Such a subclass method is said to override the superclass method, and will be selected automatically by method resolution on subclass instances. An overriding method can elect to call upon the superclass' method as part of its own implementation.

instance. (Or object instance). A specific object, as distinguished from the abstract definition of an object referred to as its class.

instance method. A method valid for a particular object.

metaclass. A class whose instances are all classes. In SOM, any class descended from SOMClass is a metaclass. The methods of a metaclass are sometimes called "class" methods (Smalltalk) or "factory" methods (Objective-C).

method. One of the units that makes up the behavior of an object. A method is a combination of a function and a name, such that many different functions can have the same name. Which function the name refers to at any point in time depends on the object that is to execute the method and is the subject of method resolution. Functions that have a common method name should also share a common *signature*.

method resolution. The process of selecting a particular function, given a method name and an object instance. The process results in selecting the particular function that implements the abstract method in a way appropriate for the designated object. SOM supports a variety of method-resolution mechanisms.

object. The elements of data and function that programs create, manipulate, pass as arguments, and so forth. An object is a way of associating specific data values with a specific set of named functions (called *methods*) for a period of time (referred to as the *lifetime* of the object). The data values of an object are referred to as its *state*. In SOM, objects are created by other objects called *classes*. The specification of what comprises the set of functions and data elements that make up an object is referred to as the *definition* of a class.

SOM objects offer a high degree of *encapsulation*. This property permits many aspects of the implementation of an object to change without affecting client programs that depend on the object's behavior.

object definition. See *class*.

object instance. See *instance*.

signature. The collection of types associated with a method (the type of its return value, if any, as well as the number, order, and type of each of its arguments). All SOM methods that have the same name should also have the same signature.

static method. Any method that can be accessed through SOM offset resolution.

subclass. A class that inherits from another class. See *inheritance*.

superclass. A class from which another class inherits. See *inheritance*.

Index

A

access classes or subclasses 1-3
access instance variables 3-9
access public instance variables 3-6
add a class object 6-61
add static method 6-1
ancestor, include section parameter 2-4
ANSI C vii, 2-12, 2-14, 2-15, 4-1
API, general usage 3-13
apply stub 6-22, 6-9
apply stub, method used to obtain 6-22
arguments, DLL 4-2

B

before or after, passthru section parameter 2-12
benefits of object-oriented programming 1-1
binding files 2-11
binding files, generating 2-1
bindings, expanding SOM macro 3-5
bindings, file types used for SOM 3-1

C

C language file types 2-2
character output function 4-3
characteristics, SOM 1-2
check version information 6-3
class 1-4, 1-2, 1-5, 3-10, 6-1, X-1
  changes to 1-2
  creating 4-3
  defining 2-1, 3-4
  exporting 1-3
  implementing 2-17
  importing 1-4
  method. See constructor
class compatibility, method used to check for 6-3
class construction process 1-5
class data, method used to get 6-25
class data, set pointer to 6-66
class definition 2-1
class file, obtain name of 6-55
class implementer API 3-10
class instance data, obtain size of 6-32
class names, identify 3-13
class name, obtain a 6-38
class object name, get 6-28
class object, create 3-6
class object, method used to find 6-5, 6-13
class object, refer to 3-7
class object, remove 6-72
class prefix, class section parameter 2-6
class readiness, method used to indicate 6-6
class registry information, transfer 6-56
classinit, class section parameter 2-6
class, data section variable 2-13
class, defining a 3-4
class, dynamically load a 6-53
class, methods section parameter 2-16
class, SOMClass 5-2
class, SOMClassMgr 5-4
class, SOMObject 5-1
clients, class 3-6
command syntax 2-20
Common Lisp Object System (CLOS) 6-10
constructor 1-5, 2-16, 2-23, 3-8, X-1
conventions 3-8
create a class instance 3-6
create a new object instance 6-58, 6-62
create class object 3-6
create new classes 1-4
creating instances 3-2
customizing SOM's use of OS/2 4-1
C++ 1-1, 1-3

D

dealing with method name collisions 3-3
debugging facilities 3-14
declaration, data section variable 2-12
declaring object variables 3-2
defining a class 3-4
description1, data section parameter 2-12
description2, data section parameter 2-13
description2, methods section parameter 2-13, 2-15
description3, methods section parameter 2-16
description4, methods section parameter 2-16
description, class section parameter 2-7
description, metaclass section parameter 2-9
description, methods section parameter 2-15
description, parent class section parameter 2-10
description, passthru section parameter 2-12
description, writing a receiving object 6-11
descriptors, identify 3-13
determine object instance class 6-49, 6-51
development process 2-1
direct access to existing classes 1-4
dispatch mechanism 6-9
DLL arguments 4-2
DLL management 4-2
DosFreeModule 4-2
DosLoadModule 4-3
dynamic typing system 6-64
dynamically load a class 6-53
dynamically loaded object, free a 6-71

E

EMITCSC.EXE 2-17
EMITC.EXE 2-17, 2-21
EMITDEF.EXE 2-17, 2-19, 2-21
EMITH.EXE 2-17, 2-19, 2-21
EMITIH.EXE 2-17, 2-19, 2-21
EMITPH.EXE 2-17, 2-19, 2-21
EMITPSC.EXE 2-17
EMITSC.EXE 2-17, 2-19
emitters 2-17, 2-19
emitters, C language 2-17
encapsulation 1-3, X-1
environment variables 2-19, 2-20
environment, run-time 1-4
error handling 3-16, 4-4
error-handling function 4-4
expanding SOM binding macros 3-5
export classes 1-3
exports statements 2-18
extended class specification 2-1
external prefix, class section parameter 2-6
external stem, class section parameter 2-5
external, methods section parameter 2-15

F

factory method. See constructor
file stem, class section parameter 2-5
file types, output 2-17
fileName argument (DLL) 4-3
find class object 6-5, 6-13, 6-15
find method procedure 6-17, 6-19
free class object 6-71
free object 6-21
function prefix, class section parameter 2-5
functionName argument (DLL) 4-3

G

generate binding files 2-1
get a class method table 6-26
get a class name 6-38
get a method offset 6-36
get apply stub 6-22
get class file name 6-55
get class instance data 6-30
get class object name 6-28
get global class data 6-25
get instance size 6-34
get name of initialization function 6-29
get number of available methods 6-39
get number of available static methods 6-40
get object class pointer 6-24
get pointer to parent class 6-41
get pointer to parent class method table 6-42
get size of an object 6-44
get size of class instance data 6-32
global class data, obtain 6-25
global or local, class section parameter 2-6

I

IBM C Set/2 Compiler 2-21
ID manipulation 3-13
identify class names 3-13
identify descriptors 3-13
identify method names 3-13
identify receiving object 6-60
IDs 3-13, 3-14, 6-5, X-1
implementation binding file 3-1
implementation conventions for methods 3-10
implementation header 2-17
implementers, API for class 3-10
import existing classes 1-3
indicate class readiness 6-6
indicate emitter programs to be run 2-19
inheritance 1-2, 2-3, 3-10, 6-36, 6-64, X-1
initialization function, obtain name of 6-29
initialize instance data 6-45
initialize new object 6-47
instance class, determine an object's 6-49, 6-51
instance data, initialize 6-45
instance data, obtain class 6-30
instance methods versus class methods 1-6
instance size, obtain an 6-34
instance variable macros, private 3-11
instance variables, accessing public 3-6
instance variables, private 2-12
instance variables, public 2-12, 3-5, 3-9
instances, creating 3-2
instance, create a class 3-6
invoke a method, dispatch mechanism 6-9
invoke any static method, macros used to 3-8
invoking methods 3-2
invoking methods by name 3-3

L

language-independent class specification 2-1
language, passthru section parameter 2-11
line, passthru section parameter 2-12
load a class dynamically 6-53
local, methods section parameter 2-15
locate included class definitions 2-19

M

macros, method 3-7
macros, parent method 3-11
macros, usage 3-6
macro, <className> New macro 3-6
macro, <className> NewClass 3-6
macro, <className> Renew 3-6
macro, _<className> 3-7

major version, class section parameter 2-6
majorVersion argument (DLL) 4-3
manipulating IDs 3-13
memory management functions 4-1
meta symbols 2-3
metaclass 1-5, 2-2, 5-2, 6-6, A-2, X-1
metaclass, include section parameter 2-4
method invocation, SOM bindings 3-5
method name collisions 3-3
method names, identify 3-13
method name, methods section parameter 2-16
method offset, obtain a 6-36
method procedure, finding a 6-17, 6-19
method prototype, methods section parameter 2-15
method resolution 1-2, 3-11, X-1
  dispatch function resolution 1-3, 6-9, 6-39
  name resolution 1-3, 2-14, 3-3
  offset resolution 1-3, 2-14, 2-16
method table, get a class 6-26
methods supported, determine an object's 6-64, 6-67
methods, implementation conventions for 3-10
methods, invoking 3-2
methods, invoking by name 3-3
methods, obtain number of available 6-39
method, methods section parameter 2-16
minor version, class section parameter 2-6
minorVersion argument, DLL 4-3
multi-threading 1-4

N

name collisions 3-3
name lookup, methods section parameter 2-16
name resolution 1-3
name, metaclass section parameter 2-9
name, methods section parameter 2-15
name, obtain class object 6-28
name, parent class section parameter 2-10
name, release order section parameter 2-8

O

object class pointer, get an 6-24
object instance, create a new 6-58, 6-62
Object Interface Definition Language (OIDL) 2-2
  class data 2-9, 2-12
  class methods 2-9, 2-13, 2-14, 2-15, 2-16, 3-8
  Class section 2-2, 2-5
  classinit 2-5, 2-6
  classprefix 2-5, 2-6
  comment styles 2-7, 2-21
  Data section 2-2, 2-9, 2-12
  definition 2-17
  external 2-14
  external prefix 2-5, 2-6, 2-15
  external stem 2-5
  file stem 2-5, 2-9, 2-10
  function prefix 2-5, 2-15
Object Interface Definition Language (OIDL) *(continued)*
  Include section 2-2, 2-3, 2-9
  internal 2-12
  local 2-14, 2-15
  major version 2-5, 2-6, 2-9, 2-10
  Metaclass section 2-2, 2-9, 2-13, 2-16
  method 2-14, 2-15
  Methods section 2-2, 2-13, 2-14
  minor version 2-5, 2-6, 2-9, 2-10
  name lookup 2-14, 2-16
  offset 2-14, 2-16
  override 2-14
  Parent Class section 2-2, 2-10
  Passthru section 2-2, 2-11
  private 2-12, 2-14, 2-15
  procedure 2-14, 2-15, 2-16
  public 2-12, 2-14, 2-15
  Release Order section 2-2, 2-8, 2-18
  sample class definition file 2-22
object variables, declaring 3-2
object-oriented programming, benefits of 1-1
Objective-C vii, 1-3, X-1
objects, run-time registry for 5-4
object, determine instance class 6-49, 6-51
object, initialize newly created 6-47
object, obtain size of an 6-44
object, release resources of an 6-69
offset resolution 1-3
offset, methods section parameter 2-16
options, SOM Compiler 2-20
override parent class method 3-11
override parent class, method used to 6-59
override parent static method 6-1

P

parent class method table, get pointer to 6-42
parent class, method used to obtain 6-59
parent class, obtain pointer to 6-41
parent method macros 3-11
parent, include section parameter 2-4
polymorphism 1-1, 1-3
preprocessor, OIDL 2-3
private instance variable macros 3-11
private-usage binding file 3-1
private, data section parameter 2-12
private, methods section paramter 2-15
procedure, methods section parameter 2-16
programs, client 3-6
public definition, SOM class 2-1
public instance variables 3-5
public instance variables, accessing 3-6
public-usage binding file 3-1
public, data section variable 2-12
public, methods section parameter 2-15

R

receiving class, test derivation 6-7
receiving object description, writing a 6-11
receiving object state, writing internal 6-12
receiving object, identification of 6-60
refer to class object 3-7
register a class object 6-61
release order 2-8
release receiving object 6-69
release storage, method used to 6-21
remove class object 6-72
reuse 1-1
root class 5-2
root class, all SOM classes 5-1
run-time environment 1-4
run-time environment, illustration of 1-5
run-time registry for class objects 5-4
running the SOM Compiler 2-19

S

SC.EXE 2-17, 2-20
set pointer to class data 6-66
side-effects 3-7, 3-8
signatures 3-3, X-1
Smalltalk 1-1, 1-3, X-1
SMEMIT environment variable 2-19, 2-20
SMINCLUDE environment variable 2-19
SMTMP environment variable 2-19
SOM bindings 1-3, 1-4, 2-1, 3-1—3-17, 6-58, 6-59, 6-61, 6-62
SOM characteristics 1-2
SOM Compiler 2-1, 2-17, 3-1
SOM functions
  somBeginPersistentIds 3-13, 3-14
  somEndPersistentIds 3-13, 3-14
  somEnvironmentNew 3-14, 3-17, A-2
  somPrintf 3-15, A-1
  somRegisterId 3-13, 3-14
  somResolve 3-5
  somSetExpectedIds 3-13, 3-14
  somTotalRegIds 3-13, 3-14
  somUniqueKey 3-13
SOM macros
  get_<instanceVariableName> 3-9
  SOM_Assert 3-14, 3-15
  SOM_CheckId 3-13, A-1
  SOM_CompareIds 3-13
  SOM_Error 3-15, 3-16
  SOM_Expect 3-14, 3-15
  SOM_GetClass 3-3, 3-16, 6-24
  SOM_IdFromString 3-3, 3-13
  SOM_NoTest 3-15
  SOM_NoTrace 3-5
  SOM_ParentResolve 3-11
  SOM_Resolve 3-8
  SOM_ResolveNoCheck 3-8
SOM macros *(continued)*
  SOM_StringFromId 3-13
  SOM_Test 3-16
  SOM_TestC 3-14, 3-15
  SOM_WarnMsg 3-14, 3-15
  <className>GetData 2-18, 3-10, 3-11
  <className>MethodDebug 3-14
  <className>New 3-6, 3-2, 3-17, 6-58
  <className>NewClass 3-6, 3-17, 4-3
  <className>Renew 3-6, 3-2, 3-17, 6-62
  _<className> 3-7
  _<methodName> 3-7, 3-3
SOM methods
  somAddStaticMethod 6-1, 5-2, A-1
  somCheckVersion 6-3, 5-3
  somClassFromId 6-5, 5-4
  somClassReady 6-6, 5-2, 6-61
  somDescendedFrom 6-7, 5-3, A-1
  somDispatchA 6-9, 5-1
  somDispatchD 6-9, 5-1
  somDispatchL 6-9, 5-1
  somDispatchV 6-9, 5-1
  somDumpSelf 6-11, 5-1
  somDumpSelfInt 6-12, 5-1
  somFindClass 6-13, 5-4, A-1
  somFindClsInFile 6-15, 5-4
  somFindMethod 6-17, 5-3
  somFindMethodOk 6-19, 3-3, 5-3, A-1
  somFree 6-21, 5-1, 6-69
  somGetApplyStub 6-22, 5-2
  somGetClass 6-24, 5-1
  somGetClassData 6-25, 5-2
  somGetClassMtab 6-26, 5-2
  somGetClassName 6-28, 5-1
  somGetInitFunction 6-29, 5-4
  somGetInstanceOffset 6-30, 5-2
  somGetInstancePartSize 6-32, 5-2
  somGetInstanceSize 6-34, 3-2, 5-2
  somGetMethodOffset 6-36, 5-2
  somGetName 6-38, 5-2
  somGetNumMethods 6-39, 5-2
  somGetNumStaticMethods 6-40, 5-2
  somGetParent 6-41, 5-2
  somGetPClsMtab 6-42, 5-2
  somGetSize 6-44, 5-1
  somInit 6-45, 5-1
  somInitClass 6-47, 5-2
  somIsA 6-49, 5-1
  somIsInstanceOf 6-51, 5-1
  somLoadClassFile 6-53, 5-4
  somLocateClassFile 6-55, 5-4
  somMergeInto 6-56, 5-4
  somNew 6-58, 1-6, 3-6, 3-7, 5-2, 6-21
  somOverrideSMethod 6-59, 5-2, A-2
  somPrintSelf 6-60, 5-1
  somRegisterClass 6-61, 5-4
  somRenew 6-62, 1-6, 5-2, 6-21
  somRespondsTo 6-64, 5-1

SOM methods *(continued)*
somSetClassData 6-66, 5-3
somSupportsMethod 6-67, 5-3
somUninit 6-69, 5-1, 6-21, 6-45
somUnloadClassFile 6-71, 5-4, A-2
somUnregisterClass 6-72, 5-4
somBeginPersistentIds 3-13
SOMCalloc 4-1
SOMClass 5-2, 1-5, 1-6, 3-4, 5-1, 5-4
SOMClassInitFuncName 4-1, 4-2, 4-3, 6-29
SOMClassMgr 5-4, 1-5, 1-6, 3-17
SOMClassMgrObject 5-4, 1-6, 2-18, 3-17
SOMClassMgr, methods used
somClassFromId 6-5
somFindClass 6-13
somFindClsInFile 6-15
somGetInitFunction 6-29
somLoadClassFile 6-53
somLocateClassFile 6-55
somMergeInto 6-56
somRegisterClass 6-61
somUnloadClassFile 6-71
somUnregisterClass 6-72
SOMClass, methods used
somAddStaticMethod 6-1
somCheckVersion 6-3
somClassReady 6-6
somDescendedFrom 6-7
somFindMethodOk 6-19
somGetApplyStub 6-22
somGetClassData 6-25
somGetClassMtab 6-26
somGetInstanceOffset 6-30
somGetInstancePartSize 6-32
somGetInstanceSize 6-34
somGetMethodOffset 6-36
somGetName 6-38
somGetNumMethods 6-39
somGetNumStaticMethods 6-40
somGetParent 6-41
somGetPClsMtab 6-42
somInitClass 6-47
somNew 6-58
somOverrideSMethod 6-59
somRenew 6-62
somSetClassData 6-66
somSupportsMethod 6-67
SOMDeleteModule 4-1, 4-2
somEndPersistentIds 3-13
SOMError 4-1, 4-4, 6-58, 6-71
SOMFree 4-1
SOMInitModule function 2-18, 4-2, 4-3, 6-29
SOMLoadModule 4-1, 4-2
SOMMalloc 4-1
SOMObject 5-1, 1-5, 1-6, 3-4, 5-2, 5-4, 6-9
SOMObject class 5-1
SOMObject, methods used
somDispatchX 6-9
somDumpSelf 6-11
somDumpSelfInt 6-12
somFree 6-21
somGetClass 6-24
somGetClassName 6-28
somGetSize 6-44
somInit 6-45
somIsA 6-49
somIsInstanceOf 6-51
somPrintSelf 6-60
somRespondsTo 6-64
somUninit 6-69
SOMOutCharRoutine 3-14, 4-1, 4-3, 6-11, 6-12, 6-60
SOMRealloc 4-1
somRegisterId 3-13
somSelf 3-10
somSetExpectedIds 3-13
somThis 3-4, 3-10
somTotalRegIds 3-13
somUniqueKey 3-13
SOM.DLL 1-6
SOM.H 2-17, 3-13
SOM_AssertLevel 3-14
SOM_CheckId 3-13
SOM_CompareIds 3-13
SOM_IdFromString 3-13
SOM_StringFromID 3-13
SOM_TraceLevel 3-4, 3-14
SOM_WarnLevel 3-14
SPC.EXE 2-17
specify directory for SOM output files 2-19
SPP2.EXE 2-17
SPP.EXE 2-17
static methods, macros used to invoke 3-8
static methods, obtain number of available 6-40
stderr 4-3
suffix, passthru section parameter 2-11
syntax, SOM Compiler 2-20
system linkage 1-6, 2-18, 4-1

T

test class derivation, method used to 6-7
transfer class registry information 6-56

U

unload class object 6-71
use, methods section parameter 2-15

V

validity-checking method calls 3-15
variable-argument methods 3-8, 6-10
VECTOR.CSC 2-22, 3-7, 3-10
version numbers 3-6, 4-3, 6-3, 6-15

virtual function 1-3

W

write receiving object description 6-11
write receiving object state 6-12

Special Characters

.C binding file 2-11
.C output file 2-17
.CS2 binding file 2-11
.CS2 output file 2-18
.DEF output file 2-18
.H binding file 2-11, 3-1
.IH binding file 2-11, 3-1
.IH output file 2-17
.PH binding file 2-11, 3-1
.PH output file 2-18
.PSC binding file 2-11
.PSC output file 2-18
.SC binding file 2-11
.SC output file 2-18
<className> ClassData 3-2, 6-25, 6-36, 6-66
<className> New macro 3-6
<className> NewClass function 2-18, 6-29, 6-47, 6-59, 6-61
<className> NewClass macro 3-6
<className> Renew macro 3-6
_<className> macro 3-7

® IBM, OS/2 and Operating System/2 are
registered trademarks of
International Business Machines Corporation

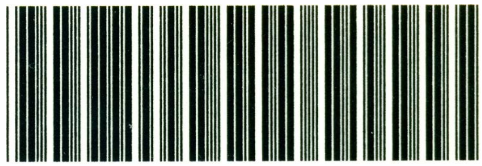
© IBM Corp. 1992

International Business
Machines Corporation

Printed in the
United States of America

All Rights Reserved

10G6309



S10G-6309-00

P10G6309